1

AN ADA-BASED FRAMEWORK FOR AN IDEF$_0$ CASE TOOL USING THE X WINDOW SYSTEM

THESIS

Jay-Evan J. Tevis II
Captain, USAF

AFIT/GCS/ENG/90D-15

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

(1)

AFIT/GCS/ENG/90D-15

AN ADA-BASED FRAMEWORK FOR AN IDEF$_0$ CASE TOOL USING THE X WINDOW SYSTEM

THESIS

Jay-Evan J. Tevis II
Captain, USAF

AFIT/GCS/ENG/90D-15

AFIT/GCS/ENG/90D-15

AN ADA-BASED FRAMEWORK FOR AN IDEF$_0$ CASE TOOL USING THE X WINDOW SYSTEM

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Systems

Jay-Evan J. Tevis II, B.S. in Computer Science

Captain, USAF

December, 1990

Preface

The purpose of this thesis is to document the design strategy, implementation methodology, testing, and evaluation performed in developing an Ada-based framework for SAtoolII using the X Window System. SAtoolII is an IDEF$_0$ graphical project editor and data dictionary editor. IDEF$_0$ is the ICAM Definition Method Zero graphical notation language adopted by the U.S. Air Force to produce a function model of a manufacturing system or environment (23:1-1). The Air Force Institute of Technology is conducting on-going research in the use of IDEF$_0$ in the requirements analysis phase of the software lifecycle. The development of SAtoolII is another part of AFIT's research efforts associated with the Strategic Defense Initiative Organization (SDIO) and its interest in the IDEF$_0$ language.

The SAtoolII predecessor program, an IDEF$_0$ graphical editor called SAtool, was written in the C programming language and used the window and graphics capabilities provided for Sun Microsystems workstations. SAtoolII not only enhances the IDEF$_0$ graphical editing features of SAtool, but also is written completely in Ada and operates independently of any specific window system or computer hardware. This independence is implied through the use of an Ada-based graphical support environment developed for SAtoolII, and the X Window System developed by the Massachusetts Institute of Technology (29:80). Along with the machine-independent window features, SAtoolII is designed around an abstract entity-relationship model of the IDEF$_0$ language. This abstract model, consisting of an essential model and a drawing model, was developed in earlier research at AFIT and revised during this research effort.

I extend my gratitude to several people who supported me during this research and software development effort. I first want to thank Captain Terry Kitchen with whom I was privileged to work for six months as he developed the essential model and expert system aspect of SAtoolII while I worked on the drawing model and graphical aspect. I also want to thank my advisor, Dr. Hartrum, whose ideas and insight made SAtoolII possible, and Dave Doak, whose system administration work made the use of the Ada bindings to the X Window System a reality. I especially want to thank my wife, Jackie, and my two little buddies, Miko and Yoshi, for their understanding and support during this research. I finally want to thank God who willingly offered His expert programming advice whenever I needed it.

Jay-Evan J. Tevis II

Table of Contents

Page

Page

Page

Page

List of Figures

AFIT/GCS/ENG/90D-15 Sub 0

Abstract

This thesis documents the design strategy, implementation methodology, testing, and evaluation used in developing an Ada-based framework for SAtoolII using the X Window System. SAtoolII is an IDEF$_0$ graphical project editor and data dictionary editor. IDEF$_0$ is the ICAM Definition Method Zero graphical notation language adopted by the U.S. Air Force to produce a function model of a manufacturing system or environment (23:1-1). The Air Force Institute of Technology is conducting on-going research in the use of IDEF$_0$ in the requirements analysis phase of the software lifecycle.

The thesis begins by providing background information on the X Window System, object-oriented data modeling, and graphical user interfaces. It then describes how SAtoolII was designed around an abstract entity-relationship model of the IDEF$_0$ language, an abstract model that was developed in earlier research at AFIT and broken down at that time into an essential model and a drawing model. It also describes the design of a machine-independent Ada graphical support environment which provides fundamental multi-window and graphical capabilities, while shielding an Ada application from the intricacies and distinctiveness of specific window systems.

Following the design information, the thesis describes how the SAtoolII program was implemented incrementally, by first developing and testing an autonomous essential model component, drawing model component, machine-independent Ada graphics support environment, and graphical user interface. The thesis ends by describing how these components can be used by follow-on research to build the completed SAtoolII software.

AN ADA-BASED FRAMEWORK FOR AN IDEF$_0$ CASE TOOL USING THE X WINDOW SYSTEM

I. Introduction

1.1 Background

Software requirements analysis, the first stage in the software lifecycle, is time-intensive and often misunderstood (32:54). As a solution to these problems, various graduate students at the Air Force Institute of Technology (AFIT) have worked over the past six years in developing a computer-aided requirements analysis tool. This tool, a user-friendly graphics-oriented computer program, has evolved in six years into a software product called SAtool. The symbolic methodology used by SAtool for software requirements analysis is called ICAM Definition Method Zero, or IDEF$_0$ (1:1). ICAM is the acronym for the Air Force's Integrated Computer Aided Manufacturing program.

The evolution process for SAtool began in 1984 when Thomas completed thesis work on an interactive computer program to generate data dictionaries (34:1). This tool ran on a VAX 11/780 minicomputer and supported SADT, data flow diagrams, structure charts, and computer source code. In 1986 Foley continued this work and designed a data dictionary editor called DDEDIT that ran on a Zenith Z-100 microcomputer (10:1). Also in 1986 Urscheler combined the design of the data dictionary editor with graphical requirements analysis to create a computer program. This program, called the Requirements Analysis Tool with a Data Dictionary (RADD), served as a prototype for SAtool. Written in C and implementing only a subset of the SADT symbols, RADD ran on a Sun Microsystems computer workstation and used the Sun graphics and windowing system (36.1). In 1987 Johnson extended RADD to include all the symbols included in the IDEF$_0$ methodology. He also added extensively to the program's data structures and increased the program's ability to derive data dictionary information from an IDEF$_0$ diagram. His data structure additions adapted the program's files to the standard interface database format developed concurrently in the research by Connally (6:1). Written in C and using the Sun graphics features, Johnson gave the name *SAtool* to this combined graphics editor and data dictionary program (18:1).

Two years following the completion of SAtool, a team of graduate students at AFIT put together an approach to $IDEF_0$ based on an entity-relationship model (1:2). This approach broke down $IDEF_0$ into an essential model and a drawing model. The essential model involved those parts of $IDEF_0$ that represented the semantics of the language and included such things as activities and data elements. The drawing data model comprised the graphical constructs used to represent the particular $IDEF_0$ analysis such as boxes and line segments. One member of this team, Smith, applied one of these models to the structure of SAtool (31:2-7). Using object-oriented design and the Ada language, Smith developed subprogram modules that implemented the essential model for SAtool. These modules operated separately from the actual SAtool program. Smith also laid the groundwork to convert the user interface of SAtool to the X Window System (31:6-2).

1.2 Problem

SAtool has fulfilled many of its expectations over the past two years, but users and researchers at AFIT have suggested several necessary changes to the program (10:5-12). The team developing the essential model and drawing model of $IDEF_0$ identified the need to change the program to conform to a more object-oriented design (1:1). Regular users of SAtool pointed out the reliance of SAtool on the windows and graphics of the Sun workstation and the need to convert the user interface to the device-independent X Window System. Shortcomings in the friendliness of the user interface, in the ability to modify already-existing diagrams, and in the creation of a comprehensive data dictionary, encouraged both users and researchers to recommend an upgrade of the entire program (18:5-12). The purpose of this research was not only to make these necessary changes to SAtool, but also to develop an Ada-based framework for SAtoolII using the X Window System.

This re-engineering of SAtool into a production-quality program resulted in the implementation of the graphical user interface and $IDEF_0$ drawing data model components of SAtoolII.

1.3 Summary of Current Knowledge

Two of the main changes suggested by the users of SAtool were to convert the user interface and graphics to the X Window System and to redesign SAtool based on an essential model and a drawing model. The first section below summarizes the current information on the X Window System, or X. The section gives a brief history of X, describes the client/server model, and explains the mechanism versus policy issue of the X Window System. The second section highlights the use of object-oriented data modeling techniques and graphical user interfaces.

1.3.1 The X Window System The X Window System is a device-independent network-based display management system. It implements the desktop metaphor used today in many graphical display computers (21:30). The X Window System has a very alphabetical lineage (29.80). The family originated at Stanford University as the VGTS, or V system, a primitive networked graphics windowing system. Then Digital Electronic Corporation desired a more advanced version of V and worked with Stanford University to develop W. Because of the needs of a networking and windowing project sponsored by IBM at the Massachusetts Institute of Technology, MIT acquired the W system and greatly improved its networking capabilities. The MIT programmers dubbed the improved W window system as X and have made the software available free to anyone who wants a copy (21:30).

X is based on the client/server model (26:354). This model makes it possible for X to operate independently of computer hardware. The server program runs on the user's computer and performs the actual window allocation and drawing tasks requested by the client program. The client, by definition, is an application program that can run either on the user's computer or on a computer connected to the user's machine by a network (26:354).

X provides the mechanisms for implementing windows but dictates no type of windowing policy (26:357). This type of operation contrasts with that offered by Apple's MacIntosh or Microsoft Windows. These software products require an application program to adhere to a particular interface style.

This section has summarized the development of the X Window System. Because of its easy availability, the absence of window policy, and the network-based client/server model, X has become a well-known window product. It is also one of the most popular choices in window interfaces for new software. By using X, the software is no longer bound to a specific graphical display or type of computer.

1.3.2 Object-Oriented Data Modeling Object-oriented programming evolved in the 1960's because of the need for simple, more understandable computer programs (37:2454). This method of programming attempts to design a computer program based on objects in the real world and operations belonging to those objects. An object is self-contained and has a state that is changed only by specific operations defined on it (35:231). This concept of object-oriented programming has given birth to subprogram modules that software designers can reuse from one application to the next. The United States Department of Defense had this g l in mind when it required reusability as a key feature of software written in the Ada programming language (37:245).

Because of the ability of object-oriented programming to model the real world, programmers over the last few years have applied this technique to the modeling of graphical data (28.27). In the modeling of solid geometry, researchers have designed software systems that allow a user to store objects in a data base fashion. Programs then change the objects' properties or query the objects about their various properties or state (28:26). At the heart of the data base approach to object-oriented data modeling is the designation of constructs, or objects, and the identification of attributes that relate the constructs. Procedures, many times called methods, are then associated with each construct. This data base approach to object-oriented data modeling has greatly increased the flexibility and understandability of graphics animation and rendering programs. In addition, it has allowed the easy integration of geometric with nongeometric data. (28:27).

1.3.3 Graphical User Interfaces Programmers have also applied the techniques of object-oriented programming to data modeling for graphical user interfaces. The growing demand for user-friendly, easily maintained, interactive software has led designers to turn to object-oriented techniques. The use of these techniques has increased the reusability and encapsulation of software and decreased the time needed for rapid prototyping. In addition, it has encouraged the iterative development of the components of a user interface (7:255).

1.4 Assumptions

The following assumptions prevailed throughout this research:

1. Johnson's work in implementing SADT symbols in SAtool was complete with respect to those symbols which could be graphically implemented (18).
2. Smith's Ada subprograms implementing the essential data model for SAtool were correct and complete (31).
3. The Ada interface source code supplied by Science Applications International Corporation (SAIC) works correctly and completely when used for calling X Window library functions from an Ada program (15).
4. The functions implemented in the X Window programming library perform as described in the X Window documentation (25).

1.5 Scope

This research concentrated specifically on the graphics and drawing modeling changes to SAtool and their relationship to Ada and the X Window System. It did not include the essential model implementation done by Kitchen in concurrent research at AFIT (20:1-1), although it did incorporate his Ada source code for the essential model into the drawing data model testing and demonstration software.

1.6 Standards

The program source code documentation used throughout the development of SAtoolII complies with the guidelines and standards written for AFIT by Dr. Thomas Hartrum (13:1). The actual Ada coding practices used were object-oriented design, loosely-coupled packages, consistent indentation, and consistent naming of packages, procedures, functions, and variable names. The application of the *use* clause for package names was avoided whenever possible to allow easier program maintenance.

1.7 Approach

The six-phase research effort to develop an SAtoolII graphical user interface and an $IDEF_0$ drawing data model implementation consisted of:

1. research into the X Window System, object-oriented data modeling, and graphical user interfaces.
2. rectification of the problems encountered by other AFIT students when using Ada with the X Window System.
3. revision of the essential and drawing models developed in earlier research at AFIT.
4. object-oriented design of SAtoolII based on the essential and drawing models along with the concepts of object-oriented data modeling, and graphical user interfaces.
5. incremental implementation and testing of the graphical user interface and $IDEF_0$ drawing data model components of SAtoolII.

6. partial integration of the SAtoolII graphical user interface, the IDEF$_0$ essential data model, and the IDEF$_0$ drawing data model source code into a text mode demonstration driver program.

In the first phase, information was collected from a variety of library sources, bookstores, and personal computer magazines.

In the second phase, the compiling, linking, and execution problems that Smith had encountered with the SAIC Ada interface code to the X Window System were remedied (31:6-3) (15:1). Most of these problems involved the pragma interface syntax in Ada and the method of linking the X Window libraries with an Ada application program. The pragma interface statement in Ada allows an Ada program to use non-Ada computer code. Refer to Appendix A for more information.

In the third phase, work in cooperation with Kitchen was done to meticulously review and revise the essential and drawing data models to ensure that all aspects of IDEF$_0$ were represented in the models (20).

In the fourth phase, more work was done with Kitchen in using an object-oriented approach to convert entity-relationship models of a system into actual Ada source code. Kitchen concentrated on the essential model while this reseach effort concentrated on the drawing model. In this phase this reseach effort also designed a machine-independent graphics support environment to provide Ada programs with the ability to utilize graphical routines and windows while shielding the programs from the intricacies and subtleties of window systems. The major impetus for the design of the graphical environment was the numerous details involved in using the X Window System and the SAIC Ada bindings to X.

In the fifth phase, this research effort implemented the drawing model entities, attributes, and relationships, while Kitchen did the same for the essential model. Both implementations involved the development of driver programs to separately test the models. Along with the drawing model, this research effort also implemented and separately tested the machine-independent Ada graphical support environment (MAGSE) and the graphical user interface for SAtoolII.

In the sixth phase, the essential model and drawing model were partially integrated along with a text mode derivative of the SAtoolII graphical user interface to form a demonstration driver program. This demonstration driver program concentrated on the box-activity relationship and the IDEF$_0$ project hierarchy of diagrams.

1.8 Equipment and Software

The following computer equipment and software were used during this research:

- Sun Microsystems Sun 3/110 workstation and AT clone personal computer
- DEC MicroVAX 3 minicomputer
- BSD UNIX operating system (version 4) and MS-DOS 3.3
- Verdix Ada compiler (version 5) and Janus Ada (version 2)
- SAIC source code modules for the Ada interface to X
- X Window System library (version 11, release 4)

The drawing data model implementation and the graphical user interface were written completely in Ada. The Ada graphical support environment (MAGSE) used by the graphical user interface was coded in Ada, with function calls made to the X Window System library by way of the Ada bindings developed by SAIC (15). Refer to Appendix D for configuration information for SAtoolII.

1.9 Sequence of Presentation

This thesis contains six chapters. Chapter I is an introduction to the thesis. Chapter II presents a literature review on the X Window System, object-oriented data modeling, and graphical user interfaces. Chapter III goes into the design specification for SAtoolII. It describes the revision of the essential model and drawing model. It also details the object-oriented design of the two models, the Ada graphical support environment, the SAtoolII graphical user interfa ·, and the integrated design approach to SAtoolII. Chapter IV covers the incremental implementation of the essential and drawing models, the graphical support environment, and the SAtoolII graphical user interfac·. It also describes how the remaining components of SAtoolII can be developed and integrated to form the complete SAtoolII software. Chapter V reviews the testing and evaluation for SAtc olII. Chapter VI presents a summary of the thesis along with conclusions and recommendations.

II. Literature Review

2.1 Introduction

2.1.1 The Need for Concise Information Many articles and advertisements today for computer programs talk about specialized window systems, object-oriented data modeling techniques, and graphical user interfaces. These programs more often than not are designed based on their own methodologies, thus contributing to the lack of understandability and portability of major software packages today. Software users and developers need concise information on how to effectively select and use object-oriented data modeling techniques, window systems, and graphical user interfaces. This literature review does just that by providing information on research and work done in each of these areas. First, this review covers the device-independent X Window System, developed by the Massachusetts Institute of Technology (MIT). Then, it describes some well-grounded object oriented data modeling techniques. Last of all, it presents advice on designing graphical user interfaces.

2.2 The X Window System

2.2.1 What is a Window System? To understand better the X Window System described in this review, the reader should first understand what a window system is, and in broader terms, what a display management system is. Stuart Lewin of Sanders Associates, Inc. provides the following definition:

> A display management system (of which the X Window System is an example) is analogous to the operating system of a general purpose computer. Display management systems provide a centralized mechanism fo· the sharing of resources between potentially competing users. In much the same way that the operating system manages access to processor cycles, peripheral devices and file systems, the display manager manages screen space, colors, fonts, cursors, and any input devices attached to a workstation. (21:30)

2.2.2 The Origin of the X Window System Most discussions of the X Window System in the literature explain where X came from. Each credits the Computer Science department at the Massachusetts Institute of Technology (MIT) as the main developer and maintainer of X; however, an article written by Robert Scheifler of MIT and Jim Gettys of Digital Equipment Corporation describes more of the details. They point out that X evolved first from 'V' and then from 'W': "the name X derives from the lineage of the system. At Stanford University, Paul Asente and Brian Reid had begun work on the W window system as an alternative to VGTS for the V system (29:80)." The writers then explain how the W window system made its way to MIT and became X:

> We acquired a UNIX-based version of W . . . produced by Asente and Chris Kent at Digital's Western Research Laboratory. From just a few days of experimentation, it was clear that a network-transparent hierarchical window system was desirable, but that restricting the system to any fixed set of application-specific modes was completely inadequate. It was also clear that, although synchronous communication was perhaps acceptable in the V system, it was completely inadequate in most other operating environments. X is our reaction to W. (29:80)

Why did MIT want a device-independent window system in the first place? The answer is simply in the portability issue. A large computer science project at MIT, called Project Athena, centered on the development of special graphical displays. For this project the programmers needed a way to run their software easily on many different types of computers (29:80).

2.2.3 Why Change To X? Readers accustomed to their own window system may ask "even though MIT had reasons to change to X, why should I spend time and effort on this new system?" IBM answers that question in its Advanced Interactive Executive (AIX) advertising brochure:

> X's popularity starts with the fact that it is in the public domain, and thus available for all vendors to use and develop in their own way. Also important is the fact that X represents an important breakthrough in distributed computing (commonly referred to as "networking"). X is particularly useful in environments where PCs, workstations, and minicomputers from different vendors need to run the same application. (16:53)

2.2.4 The Client/Server Model and Distributed Computing The client/server model of X is the basis for the distributed computing written about in IBM's AIX brochure. This model makes it possible for X to operate independently of a specific computer hardware (26:354). The server program runs on the user's computer and performs the actual window allocation and drawing tasks requested by the client program. The client program, by definition, is an application program that can run either on the user's computer or on a computer connected to the user's machine by a network (26:354). In simple terms, the client program describes a picture to draw and the server program draws the picture on the user's computer display.

A skeptic reading about the distributed computing approach might comment first on decreased performance, that is, the increased execution time as the user sees it. Scheifler and Gettys confront that issue in their article:

> The performance of existing X implementations is comparable to that of contemporary window systems and, in general, is limited by display hardware rather than network communication. For example, 19,500 characters per second and 3500 short vectors per second are possible on Digital Equipment Corporation's VAXStation-II/GPX, both locally and over a local-area network, and these figures are very close to the limits of the display hardware. (26:80)

2.2.5 Mechanism But Not Policy Along with good performance, X also provides mechanisms for implementing windows but dictates no type of window policy (26:357). A client program issues suggestions or hints on window position and size, but the server program has the final say on the actual window dimensions and its exact location. This type of operation contrasts with that offered by Apple's MacIntosh or Microsoft Windows. These software products require an application program to adhere to a particular interface style (38:2).

A by-product of mechanism over policy is the development of many window managers for X. A window manager runs on the user's computer and controls such things as how the various fonts appear in certain windows and how programs can communicate with different peripheral devices. A window manager may make up part of an operating system. It also may exist as a separate computer program (24:65). These window managers provide the interface between the user and X. Each of the managers implements the window policy preferred for that computer without having any effect on the operation of the application program (21:31)).

2.2.6 X Out On The Road Because of its mechanisms, X has proved its worth in emulating other window systems. X Window System managers now exist that appear to the user as a Macintosh or Microsoft Windows display. X window managers that emulate the OS/2 Presentation Manager are now in development by software companies (26:357).

X has also shown that it can handle multiple windows with ease. Douglas Young describes an X Window System environment in which an interactive computer training program (an X Window client) executes on the school's main computer. In a classroom each student has a personal computer with an X Window manager on his desk. Each computer display has a window for the computer training program and possibly one for checking electronic mail. On the instructor's desk is a single computer monitor showing each student's screen display (21:2–3).

2.2.7 Conclusion This literature review has introduced the reader to the X Window System. It told of the origin of X and how computer scientists at MIT developed the X Window System to fill a distributed-computing need. The review also explained the client/server model of X and how this window system approach supplies mechanism rather than policy.

2.3 Object-Oriented Data Modeling

2.3.1 The Need for Object-Oriented Data Modeling Most programmers are familiar with functional-oriented modeling. The FORTRAN programming language and the HIPO charts of COBOL are good examples. In working with this method of modeling, these same programmers will tell you about problems in building large, complex systems. Shlaer and Mellor point out the consequences of problems that occur when relying on a functional-oriented design of large systems (30:4–5):

- Floundering in Analysis. The analysts try to take everything into account.
- Requirements Failure. A requirements document is produced but no one knows enough about the whole system to spot the incompleteness or inconsistencies in it.
- Premature Rush to Implementation. The designers shrug off understanding and just precede in trying to design and implement it.
- Faults and Inconsistencies. After classifying certain program modules as independent of one another, they are many times developed by a number of programmers. The concept of independence between modules is then falsely extended to the data the modules deal with.
- Unintelligent System. The system is designed with no notion of the differences and relationships between each of its parts.

An answer to the causes of these consequences is object-oriented data modeling through the use of information models.

2.3.2 Objects, Attributes, and Relationships An informational model is based on the real world and is made up of objects (or entities), attributes of objects, and relationships between objects (30:6–13). In determining attributes, the designer should strive to have them capture all pertinent information, exist uniquely from other attributes of an object, and take on values independent of the other object attributes (30:26). Attributes fall into three categories : descriptive, naming, and referential. (30:29–32). Descriptive attributes and naming attributes have to do with only the object itself, but referential attributes deal with the relationships between objects.

Referential attributes provide the mechanism that tie two or more objects together in a relationship. These relationships can be 1-to-1, 1-to-many, or many-to-many. A 1-to-1 relationship is modeled by adding a key field attribute in one object that uniquely identifies the object it is related to (30:52). A 1-to-many relationship is modeled similarly by placing a key field attribute in the *many* object that uniquely identifies the *1* object.

Modeling the many-to-many relationship is more involved than the methods described above. The key field attributes are placed in a structure separate from any of the objects composing the many-to-many relationship. This structure is referred to as a correlation table if it contains only corresponding key field attributes (30:58), and an associative object if it also has distinct attributes of its own (30:69).

2.3.3 Data Model Implementation : The Keystone Methodology Object-oriented data modeling, also referred to as entity-relationship (E-R) modeling, does not provide for system implementation. This step is left up to the programmers and analysts. Eric Kiem served as part of a programming team that implemented a system using E-R modeling. According to Kiem, his E-R implementation methodology, called the Keystone Methodology:

> uses Entity-Relationship modeling to determine an optimum object-oriented packaging structure, which will exhibit minimum coupling and interdependencies between elements of a system and therefore maximum reusability potential. Furthermore, the resulting organization of the data dimension permits extensive use of a limited range of generics to provide complete data manipulation through the use of relational operations. The form and disposition of concurrent elements of a system can also be determined directly from the E-R model. The modeling process is proven and the implementation of the resulting design is systematic (19:101).

Kiem points out that as relationships are formed by associating entity key values, a relationship itself can become an object (19:102). This corresponds to the Shlaer and Mellor correlation table or associative object used to model many-to-many relationships. Kiem also states that E-R modeling of a system followed by the Keystone Methodology of design results in "minimum coupling and maximum reusability of all entities." (19:105)

2.3.4 Conclusion Object-oriented data modeling overcomes the problems associated with the functional modeling of large software systems. In data modeling, analysts completely identify all objects, attributes, and relationships in a system. This modeling technique takes into account all the data in the system, the objects that use that data, and the relationships between those objects. Depending on the type of relationship, each can be modeled as either a key field attribute of an object or as an entry in a correlation table or associative object. The Keystone Methodology provides one proven way of going from system model to actual implementation.

2.4 Graphical User Interfaces

2.4.1 Why A Graphical User Interface? A graphical user interface (GUI) employs multiple windows, menus, icons, and a mouse to make communicating with a computer more productive and less frustrating. This is based on research done by Microsoft Corporation and Zenith Data Systems Corporation (22:91). A GUI frees the user from burdensome manual reading or command memorization. An example of a GUI is the X Window System described earlier in this literature review. Because of the client/server model and network communication ability, the X Window System is more sophisticated than other GUIs (22:91).

2.4.2 An Object-Oriented Approach to a Graphical User Interface The object-oriented modeling described already in this literature review can be effectively applied to GUIs. An example is the Graphical Object Workbench (GROW) system developed by Paul Barth. In the design of GROW, Barth used three techniques (2:142):

- Object-based graphics based on a taxonomic hierarchy and utilizing inheritance
- Inter-object relationships which allow the composition of several objects into more complex objects which then have interdependencies

- Separation of interface and application to simplify the modification and reuse of the interface for other applications

Barth relates his graphical objects together through the use of taxonomic inheritance, composition, and graphical dependency (2:148). Taxonomic inheritance involves each kernel (the most general) graphical object in the system having attributes and methods (functions) used for graphical interaction and for displaying and moving the object. Other objects below these kernel objects either inherit these kernel attributes and methods or override them with their own (2:149). Composition allows objects to be grouped into complex objects which are then treated as if each were a single object itself. The objects are linked together by key fields called slots (2:150-151). Graphical dependency between objects ensures that when an attribute is changed, the dependent attributes are also changed. It is implemented as an attribute in an object. The attribute consists of a list of other attributes on which the object depends (2:152).

Barth sets a definite barrier between the interface and the application to reduce the complexity of interface reuse. This barrier does not permit the application to access GROW's internal data structures; all communication occurs through the use of keys that are passed to an application by GROW and uniquely identify a graphical object (2:156–157). This is analogous to the client/server model in the X Window System.

In creating the GROW interface, Barth followed four steps (2:159–163):

1. Create graphical objects to be used in the interface and define their composite structures. This involves the creation of a class object for each type of object.
2. Establish graphical dependencies. This involves linking objects through dependency attributes.
3. Make changes or extensions to the methods and attributes provided by the system. This involves specializing a method or an attribute to a object instance.

4. Link the interface and application. This involves setting up calls from the GROW interface to the application, and from the application back to GROW. The calls are mainly menu-driven.

The GROW interface developed by Barth brought out many important aspects of graphical user interface design. First, the three basic object relationships are essential to the versatility and power of the system. Second, dependencies should be defined between object attributes rather than the actual objects. Finally, separation between the interface and the application permits the interface to be used for a variety of different application needs.

An application that borrowed extensively from Barth's work with GROW is described by Paolo Sabella and Ingrid Carlbom. They incorporate object-oriented data modeling with its attributes, relationships, and inheritance into a system that manipulates solid geometric shapes (28:24).

2.4.3 Graphical User Interface Consistency A review of GUIs should not close without a discussion of consistency issues, especially with all the various graphical user interfaces available today. Jonathan Grudin argues "for a shift in perspective" because "when user interface consistency becomes our primary concern, our attention is directed away from its proper focus: users and their work." (11:1164)

Grudin lists three types of user interface consistencies:

- Internal consistency of an interface design. This receives the most attention by designers.
- External consistency of interface features with features of other interfaces familiar to the users. The big issue here is not having to retrain the users.
- Correspondence of interface features to familiar features of the world beyond computing. This involves the appearance on the screen of objects or devices that the user sees in his work area every day.

His study of user interface consistency brought out that "consistency that supports ease of learning can conflict with ease of use." (11:1167) Two items to consider in this area are the positioning of default menu selections and the abbreviations of commands or operation names (11:1169). Grudin concludes by stating that designers should strive for a user interface that tends more towards matching the user's work environment than being consistent. They should not let the user interface be driven by the underlying system architecture. In addition, the designers should recognize that a fully consistent system is not feasible (11:1170–1172).

2.5 Summary

Software users and developers need information on the use of window systems, object-oriented data modeling, and graphical user interfaces. This literature review has briefly covered all three of these areas and has given examples of their use. The next chapter in this thesis shows how information presented in this review was used in designing the machine-independent Ada graphical support environment, the SAtoolII drawing model, and the SAtoolII graphical user interface.

III. SAtoolII Design

3.1 Introduction

This chapter takes the reader through the design specification phase of SAtoolII. It starts off with the next section summarizing the SAtoolII predecessor program called SAtool and showing how information was drawn from experiences with SAtool to improve the design of SAtoolII. The third section retells the history of the essential and drawing models initially designed to model IDEF$_0$, while the fourth section explains the changes that were made to those models as a result of reexamination in this and Kitchen's research. The fifth and sixth sections describe the object-oriented design of the machine-independent Ada graphical support environment and the SAtoolII graphical user interface. The seventh section goes into the design of complex drawing objects needed in IDEF$_0$ diagrams, while the eighth section tells how these diagrams are then saved in project files. The ninth section describes an error handling design to centrally deal with all runtime errors that occur during the execution of SAtoolII and the tenth section explains why a special component is needed in the design for macro operations and project integrity constraint management. The eleventh section illustrates how each of the component parts of SAtoolII fit together in an integrated SAtoolII design specification.

3.2 SAtool in C

3.2.1 SAtool Design and Implementation The predecessor computer program for SAtoolII, called SAtool, was developed by Johnson as part of a thesis research project at AFIT in 1987 (18:1-1–1-11). He wrote the program in the C programming language and used functions calls to the Sun Microsystems SunView window and graphics libraries to implement his graphical user interface and drawing routines (33) (18:4-1–4-2). In designing and implementing SAtool, he used a top-down approach that involved first putting together the menus in the user interface and then implementing each of the functions identified in the menus. By selecting the proper menu items, and designating location points and text, a user could create, store, and reload a single IDEF$_0$ diagram (18:4-18). The structure charts in Johnson's thesis document this functional approach to the overall operation of SAtool (18:G-1–G-18).

In designing the primary data structures, Johnson took an object-oriented approach and used the following design objectives (18:4-10):

- The data structures must maintain both graphics and data dictionary information.
- The data structures must separate the data dictionary information from the graphics information as much as possible.
- The data structures must maintain enough information to allow the graphics and data dictionary information to be stored in separate files and later restored from those files.

Johnson's primary data structures consisted of a box, line, squiggle, header, and footnote. These data structures made it possible to create, save, and load both graphical and data dictionary information. However, as Johnson pointed out, SAtool could not consistently resolve the source and destination fields for a data dictionary entry from just the graphical information stored in the data structures (18:4-15). Along with the primary data structures, Johnson identified the following five graphical entities: activity box, ICOM line, squiggle line, diagram label, and footnote marker (18:5-12).

Johnson set up SAtool to create four types of data files, all in an ASCII format. The data file types are (18:4-16–4-17):

- A file with a .gph extension to store the diagram graphics information
- A file with a .dbs extension to store database information that followed the guidelines determined by Connally in (6)
- A file with a .fpt extension to store facing page text information to accompany an IDEF$_0$ diagram

- A file with a .dd extension to store a formatted version of all or parts of the contents of the .dbs file

To generate a copy of a completed IDEF$_0$ diagram, Johnson used the window capturing feature of the Sun workstation. This feature places the image contained in a Sun window into a file. The user can then send the file to a laser printer which produces a copy of the window contents on paper (18:4-17).

3.2.2 Suggested Changes to SAtool After implementing SAtool, a group of AFIT students evaluated the program's performance (18:5-8–5-9). This group gave the following suggestions (18:5-11–5-12):

- Improve the user documentation and give examples instead of making the user learn by trial and error
- Add a help selection to each of the menus
- Implement an UNDO command for each menu selection

Johnson also gave some recommendations for improvements to SAtool (18:6-2–6-4):

- Integrate the use of voice output by the software
- Change the diagram printing capability from a simple time-intensive pixel dump to a file of line and text drawing commands
- Improve the simple editing features of data dictionary input
- Provide on-line help
- Convert the user interface and graphics from Sun View to a standard graphics package such as GKS
- Enhance the software to handle an entire project instead of just one diagram at a time

- Design the software to create a diagram solely from the contents of a data dictionary
- Add color to the interface
- Redo the software in Ada

Students in the software engineering classes at AFIT have used SAtool since its creation in 1987. This period of usage has produced other recommended changes to SAtool (14:1-27):

- Have the lines and arrows connected to box automatically move when the box is moved
- Have the software automatically space lines evenly on the side of a box
- Reduce the number of menu choices needed to perform drawing tasks
- Make the software more sensitive to drawing changes
- Explain more clearly the user errors when they are made
- Automatically save the data dictionary information when the diagram is saved
- Include a tutorial and examples in the documentation
- Provide drawing and manipulation operations specifically useful for IDEF$_0$ diagrams rather than duplicating the features commonly available in any paint program
- Improve the user confidence in the system by providing more feedback and by gracefully handling user and program runtime errors
- Make the text in the diagrams easier to read
- Make the software more useful as an IDEF$_0$ creation and modifying tool rather than as just a means of fancying up a design already done on paper
- Make the menu choices and prompts duplicate the process followed in creating and modifying an IDEF$_0$ diagram on paper
- Create a version of the software to run on a PC or at least allow viewing and printing capabilities other than those on a Sun workstation

3.2.3 Some Final Thoughts on SAtool Johnson methodically approached the designed and implementation of SAtool. Although he based his primary data structures on an object-oriented design, his overall program flow and control in SAtool were purely functional. In short, the driving force behind the development of SAtool appears to have been the desired menu functions and the drawing, saving, reloading of a graphical diagram. The ability to subsequently build a data dictionary from the created drawing seems to have only been a by-product or secondary consideration of the SAtool design.

Johnson's SAtool laid the groundwork for the formulation of the essential model and drawing model and for the design and development of SAtoolII. His overall design, although functional, provided helpful information used later in drawing and manipulating the SAtoolII diagrams. The remaining sections in this chapter incorporate into the design of SAtoolII both Johnson's work and the items in the lists of recommended changes.

3.3 Original Essential and Drawing Models

3.3.1 The Original Essential and Drawing Models for IDEF$_0$ In 1989, students doing research work at AFIT created an entity-relationship or E-R model of the IDEF$_0$ language (1:1). They separated the model into an activity essential data model shown in Figure 3.1, a data element essential data model shown in Figure 3.2, an acitivity drawing data model shown in Figure 3.3, and a data element drawing data model shown in Figure 3.4 (1:3–4). According to Smith,

> ...the essential data was defined to be all the data required to compile data dictionary entries for the activities and data elements. The drawing data was defined to be any data which was not essential. The drawing data mainly consisted of data depicting the graphical layout of the diagrams and supporting text such as footnotes. Efforts were made to prevent the storage of a single datum as essential data and drawing data. As a result of our efforts, four separate ER diagrams were developed." (31:4-1–4-2)

The entities, attributes, and relationships used in these models were primarily identified from a review of Johnson's SAtool design and implementation (18), the structured analysis article by

Ross (27), and the IDEF$_0$ manual (23). The amalgamation of these models with the details of IDEF$_0$ and the operation of SAtool appear in a technical report by Hartrum (12).

3.3.2 Initial Implementation of the Essential Model As a first step in developing the next version of SAtool, Smith set out to implement the essential model part of the entity-relationship model for IDEF$_0$ by using it as a basis for an Ada implementation of SAtoolII (12:88). He ended up developing what he called an IDEF$_0$ Syntax Data Manipulator. The data manipulator allowed the user to create, modify, link together, and delete activities and data elements. It also loaded and stored data dictionary information (31:6-1–6-3). Therefore, what remained for the future was an implementation of the drawing model and an improved user interface. This implementation would then be combined with Smith's essential model implementation to form SAtoolII (12:88).

3.4 Revised Essential and Drawing Models

3.4.1 Review and Revision of the IDEF$_0$ Models Meeting first in January of 1990 and intermittently through July, this research effort set out in conjunction with Kitchen's work to review and possibly revise the essential model and drawing model diagrams. Kitchen reviewed the essential model and Smith's work with it, while this research effort analyzed the drawing model and how it related to Johnson's SAtool work and IDEF$_0$.

3.4.2 Changes to the Essential Model Kitchen found the existing design and implementation of the essential model to be inconsistent and inadequate, and suggested many changes (20). He removed the *analyzes* relationship and *analyst* entity from both the data and activity model diagrams and moved the attributes up to the *activity* entity and *data element* entity. He also completely did away with the *alias* entity. In addition, he redesigned the *consists of* relationship which was connected to the *data element* entity. The resulting revised essential model diagrams appear in Figure 3.5 and in Figure 3.6. This essential model description served as the design specification for the essential model component of SAtoolII (20).

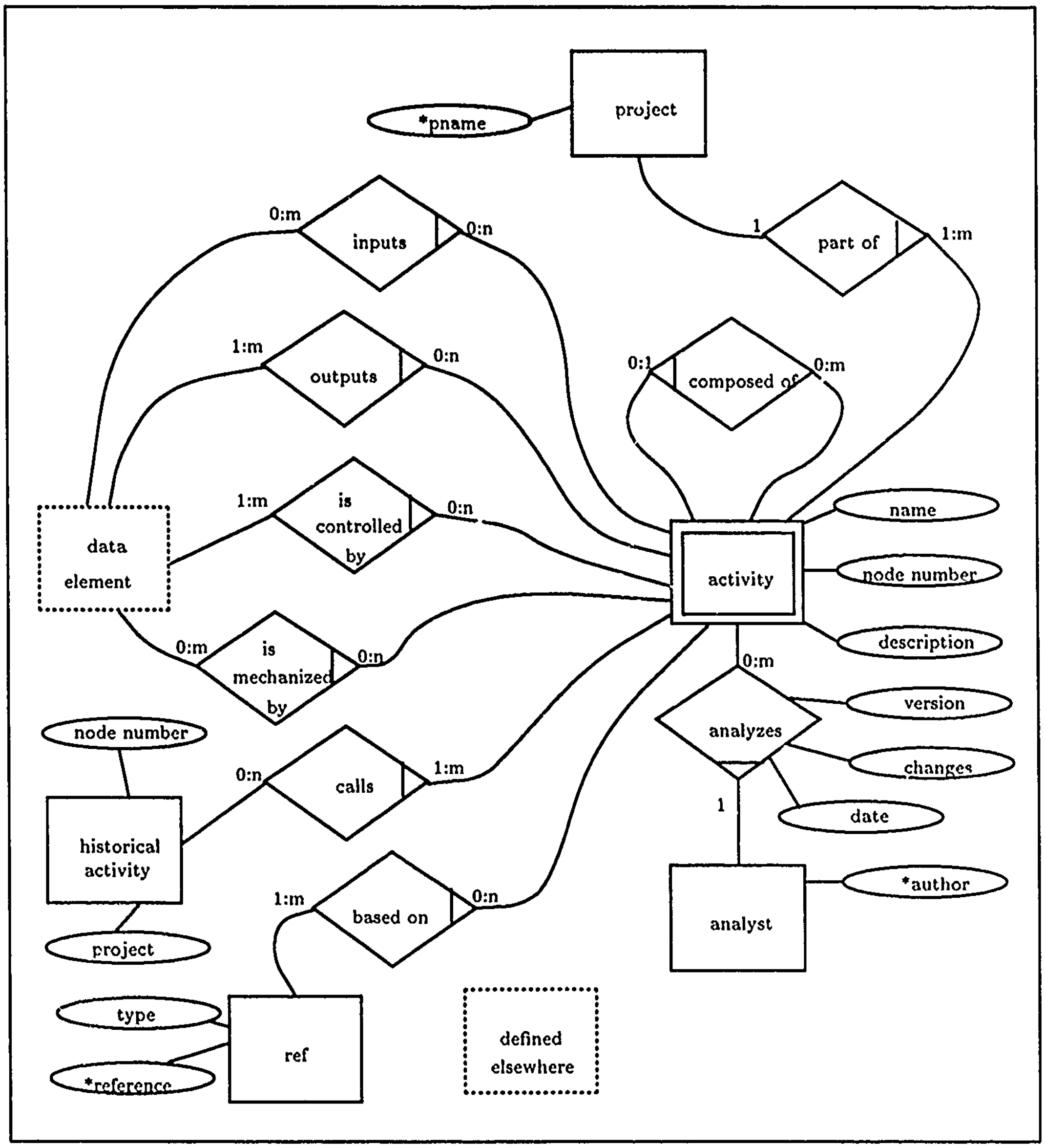


Figure 3.1. Original IDEF$_0$ ACTIVITY Essential Data Model

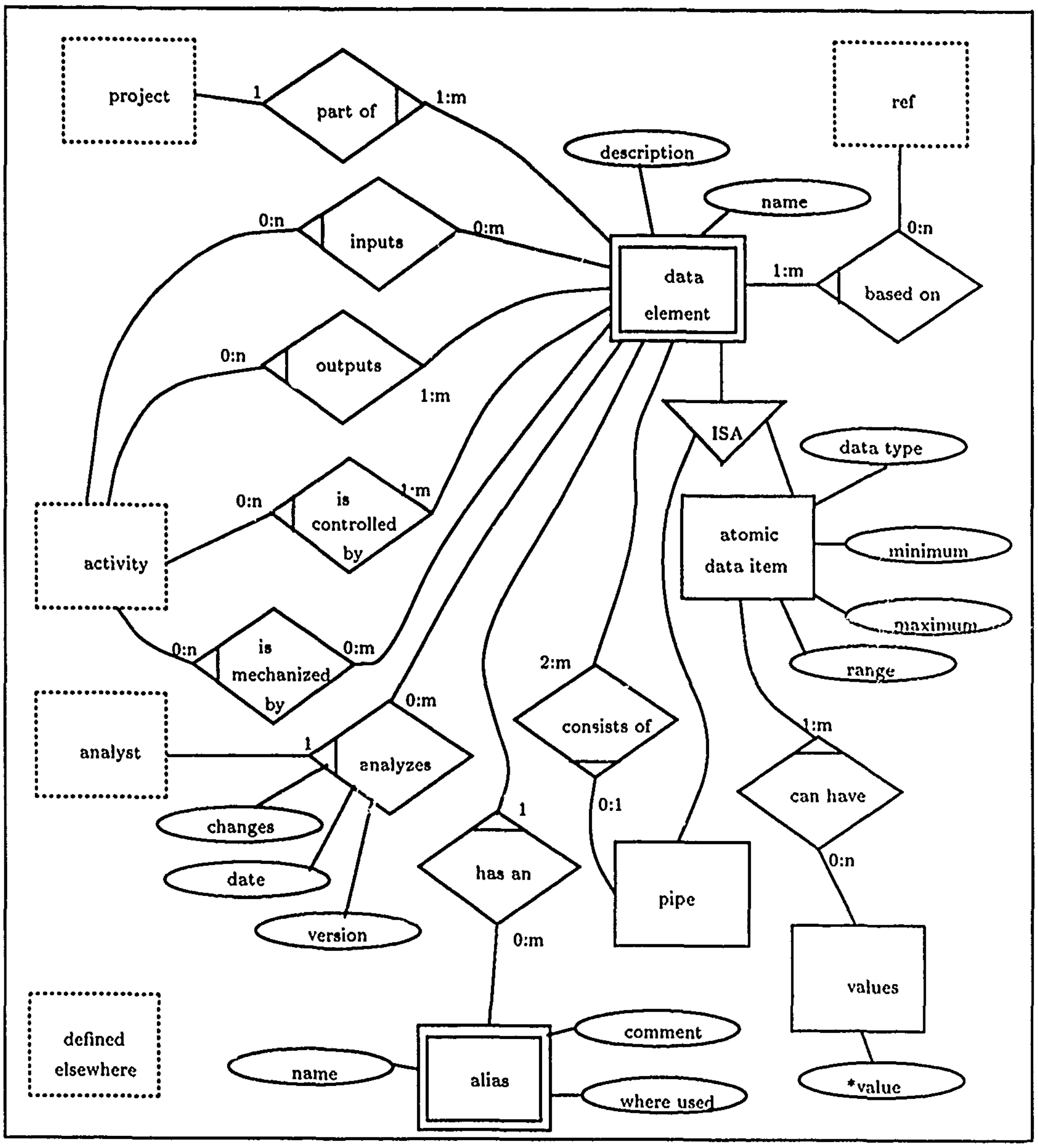


Figure 3.2. Original IDEF$_0$ DATA ELEMENT Essential Data Model

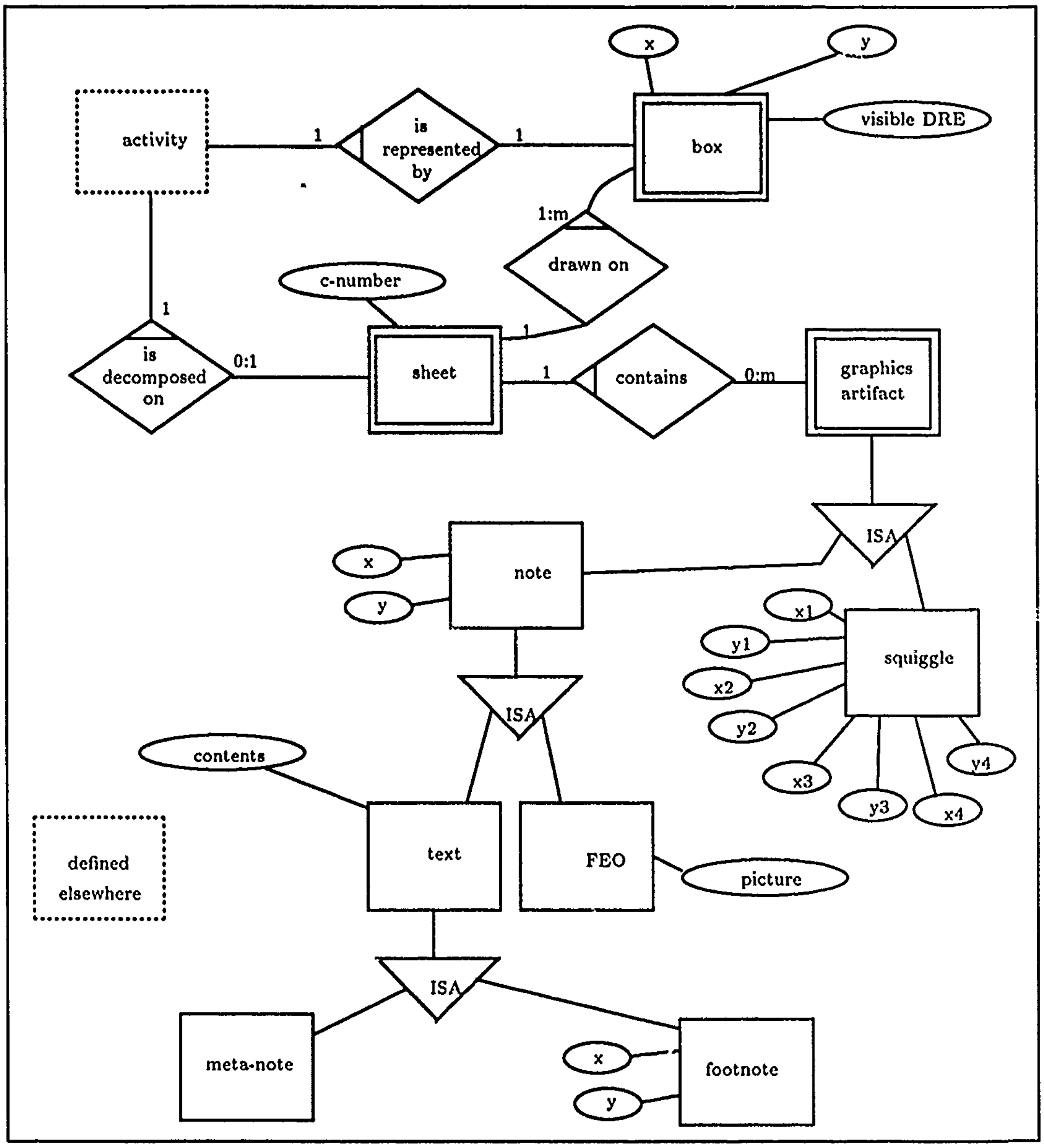


Figure 3.3. Original IDEF$_0$ ACTIVITY Drawing Data Model

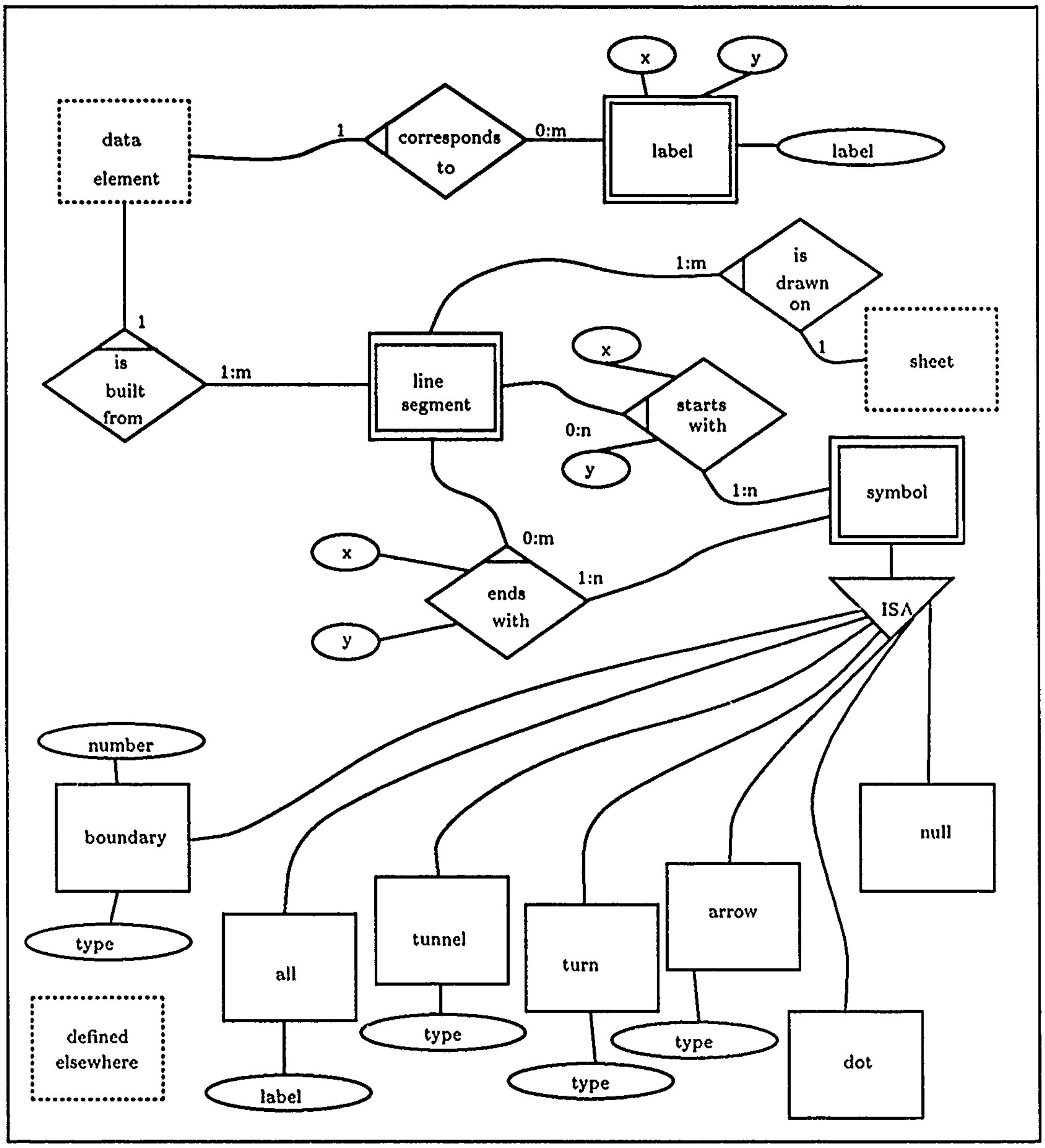


Figure 3.4. Original IDEF$_0$ DATA ELEMENT Drawing Data Model

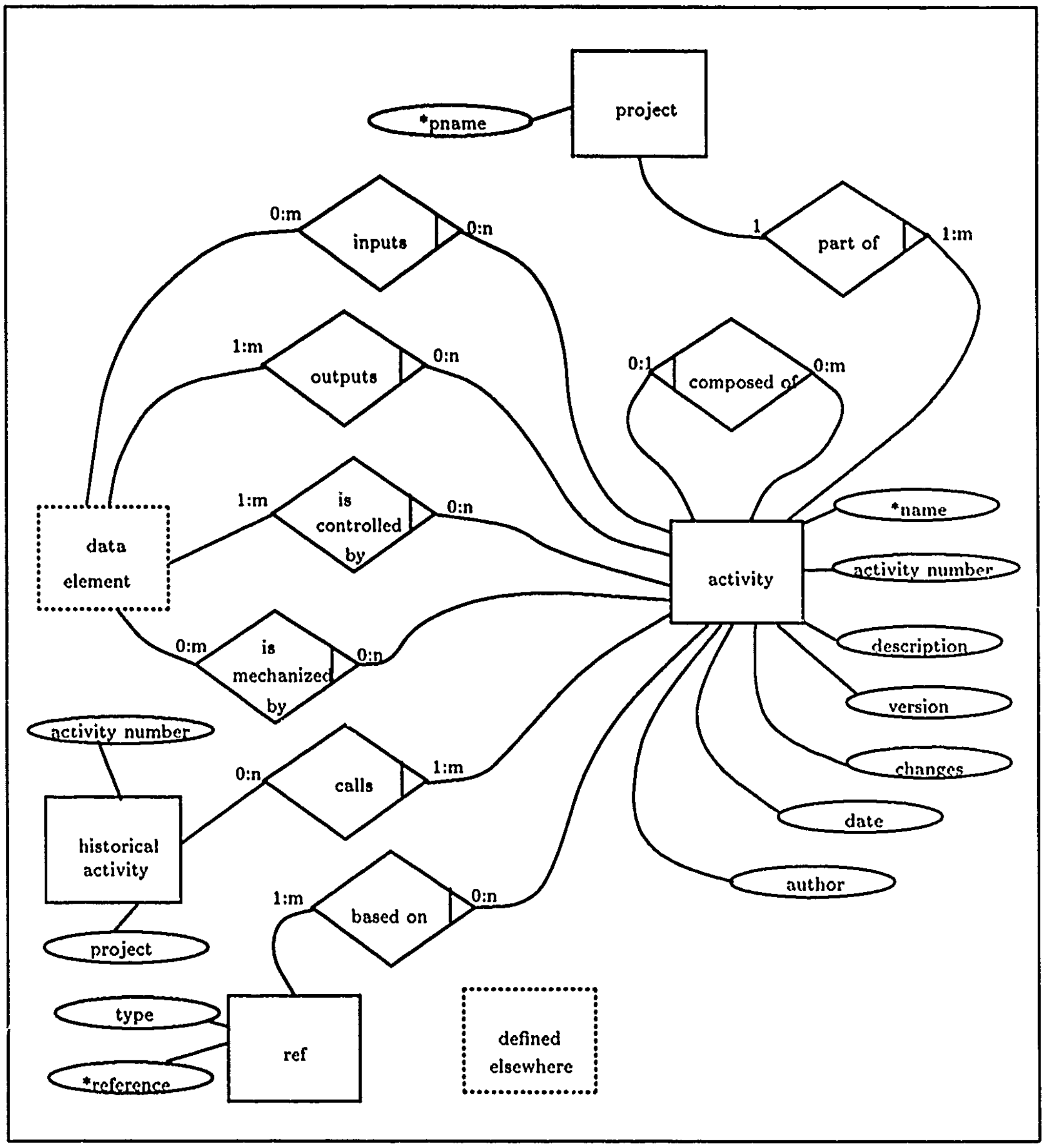


Figure 3.5. Revised IDEF$_0$ ACTIVITY Essential Data Model

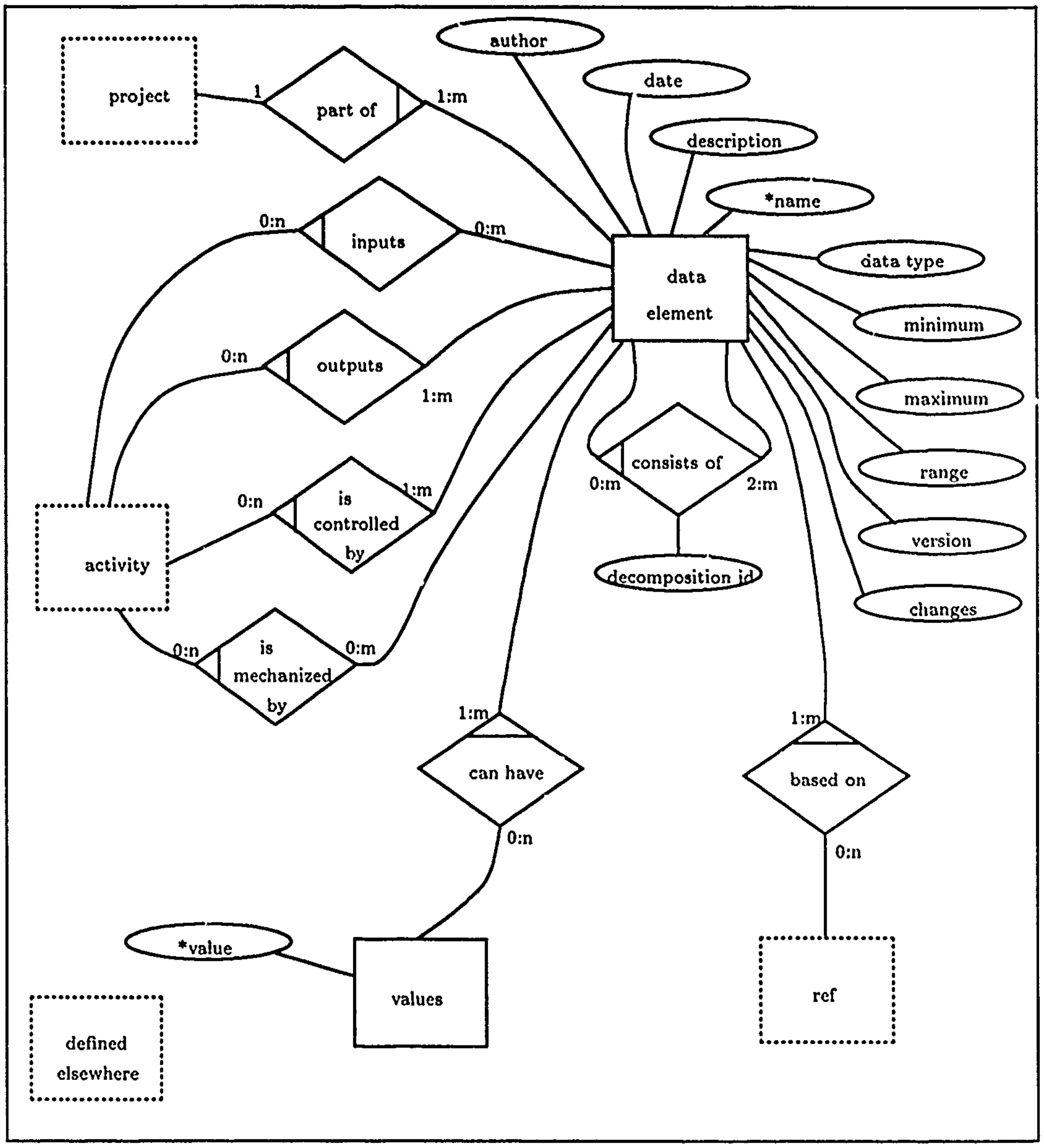


Figure 3.6. Revised IDEF$_0$ DATA ELEMENT Essential Data Model

3.4.3 Changes to the Drawing Model Upon reviewing the drawing model E-R diagrams for IDEF$_0$, this research effort spotted ambiguities and conflicting concepts in the design. The separation of the drawing model into an activity model and a data element model seemed inappropriate and almost forced upon by the design of the essential model. The drawing model actually consisted, in part, of the graphical entities that Johnson had identified in his thesis (18:5-12). Also, the corresponding drawing model entity for the essential model activity, the box, was really only one of nine other entities in the drawing model. Therefore this research effort applied the principles of object-oriented data modeling described by Shlaer and Mellor (30) and the graphical object relationships identified by Barth (2:147-148) to create a completely revised drawing model. As in the original drawing model, this drawing model uses entity, attribute, and relationship names and definitions from the notational language that IDEF$_0$ is based on (27).

3.4.4 The Revised Drawing Model The revised drawing model is divided into three figures. Figure 3.7 shows the drawing entity class and its subclass entities. Figure 3.8 shows all the entities and their attributes. Figure 3.9 shows the entities and the relationships between them.

This revised model uses the entities and attributes described below. As illustrated in Figure 3.7, most of these entities are subclasses of the drawing entity. The exceptions to this are the note, footnote, and metanote entities which are subclasses of the verbal addition entity. Each of the entities has both unique attributes and attributes of its superclass. In the case of the drawing entity, this is the entity name and the entity's X and Y position on a diagram. In the case of a verbal addition entity, this is the text or graphical contents.

- diagram - A diagram contains the decomposition of a box. Its unique attribute is a C-number, or chronological creation number. The C-number is made up of an author's initials and an integer. This notation refers to an existing diagram that will be hierarchically renumbered when the project is completed (27:32).

- box - A box represents an activity on an IDEF$_0$ diagram. Its unique attribute is a DRE, or detail reference expression. This includes such notation as C-number, page number, or SA call (27:32).
- FEO - An FEO, or for exposition only, is a special effect feature placed in a diagram. It is not part of a diagram, but is used to illustrate the purpose of a particular action taken on the diagram. Its unique attribute is a picture acting as an embedded illustration (27:23).
- footnote - A footnote is the same as a note except that it can be used instead of crowding in part of the diagram forces the text to be moved to another diagram location. Its unique attribute is an X-Y marker which holds the location in a diagram of the number referring to the footnote positioned elsewhere in the diagram.
- label - A label tells what specific information a line segment or an historical activity represents. It is used in conjunction with a squiggle if the line segment it corresponds to is not obvious. One of its unique attributes, the text attribute, contains the label characters. Its other unique attribute is a boolean flag indicating if the label is used for a historical activity or a data element.
- line segment - A line segment, in connection with other line segments and terminators, shows the flow of data from one box to the next. This data is identified by a label related to the line segment. One or more line segments connected to terminators are used to represent a data element. A line segment has no unique attributes.
- metanote - A metanote is not part of the IDEF$_0$ description in a diagram. Instead it is information about the diagram, such as the way it is laid out or the choice of label or box names. It has no unique attributes.
- note - A note puts nongraphical analysis-related information into a diagram. If the object that the note refers to is not obvious, a squiggle can be used to clarify the situation. A note has no unique attributes.

- squiggle - A squiggle is a device used when crowding on part of a diagram causes poor readability. It is used to relate a label to a line segment or a footnote marker to a line segment when the label cannot be placed close enough to the object. A label has two unique endpoint attributes, one near the label or footnote marker and one near the line segment.

- terminator - A terminator is a symbol that attaches to a connector. The other end of the connector may be attached to a line segment, a box, or another terminator. A termin identifies where the data is going and possibly something about where it came from. The terminator can have one of nine identifiers in its unique symbol attribute: arrow, boundary arrow, tunnel arrow, to-all, from-all, simple turn, junctor, dot, or null (stands for no terminator). A junctor is a three-way line segment intersection. Its unique direction attribute contains more layout information about a terminator and its unique ICOM code attribute contains a character string.

- connector - A connector is not an IDEF$_0$ object, at least not a visible one. It allows the drawing model to take the tinker toy approach in forming a diagram. It has no unique attributes.

Figure 3.10 is an illustration of how the drawing model entities work together to form an IDEF$_0$ diagram. The most prominent item in the figure is the connector stub. Although connector stubs are not visible, they provide the ability to easily move related objects around on a diagram. A single move of a connector stub can automatically relocate a number of other drawing objects. The revised drawing model description contained in the drawing model figures served as the design specification for the essential model component of SAtoolII.

3.5 Machine-Independent Ada Graphical Support Environment (MAGSE)

3.5.1 Requirements Analysis for the MAGSE

The machine-independent Ada graphical support environment or MAGSE should provide an interface between any window system and an Ada

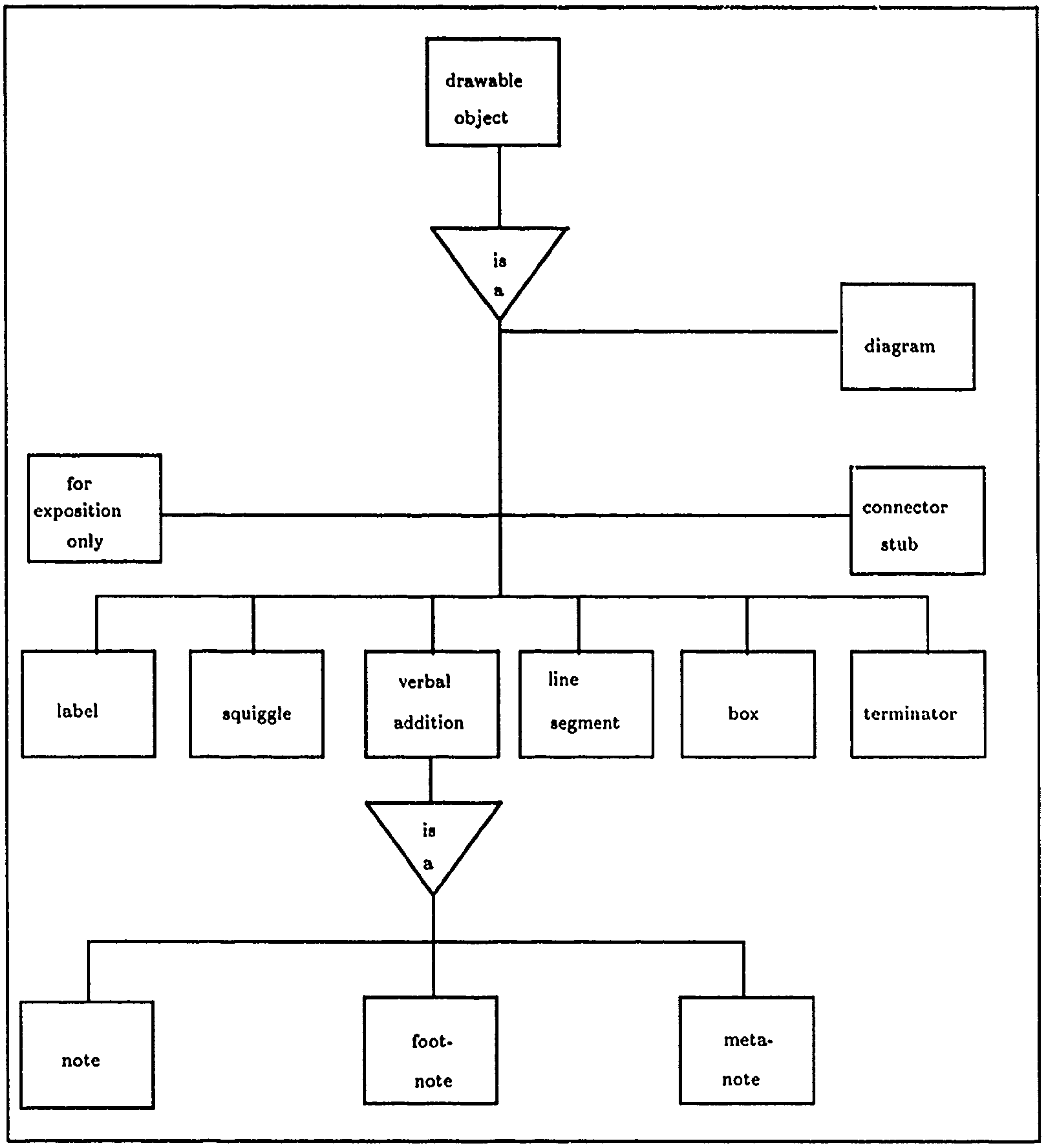


Figure 3.7. IDEF$_0$ Drawing Data Model (Classes and Subclasses)

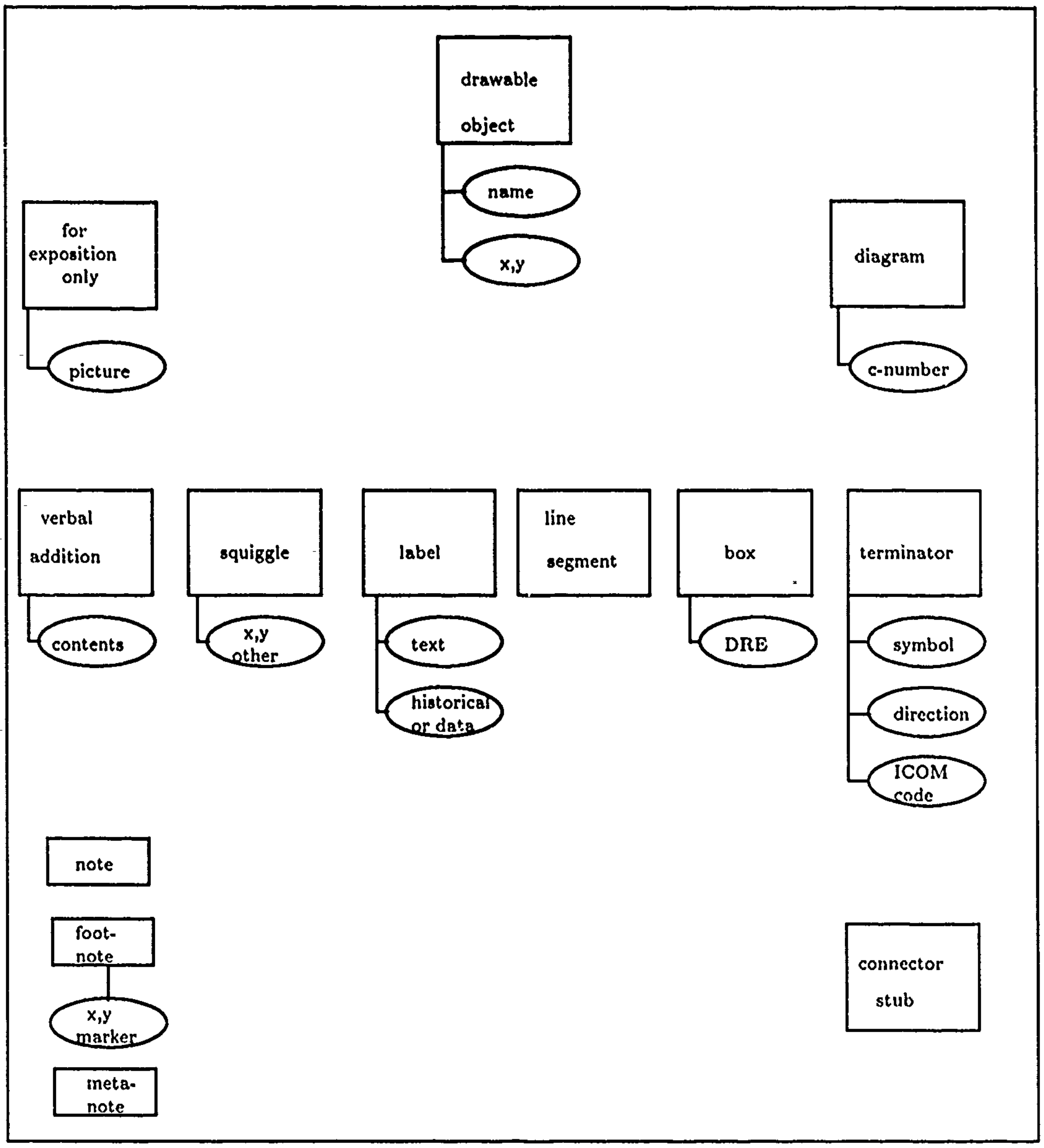


Figure 3.8. IDEF$_0$ Drawing Data Model (Entities and Attributes)

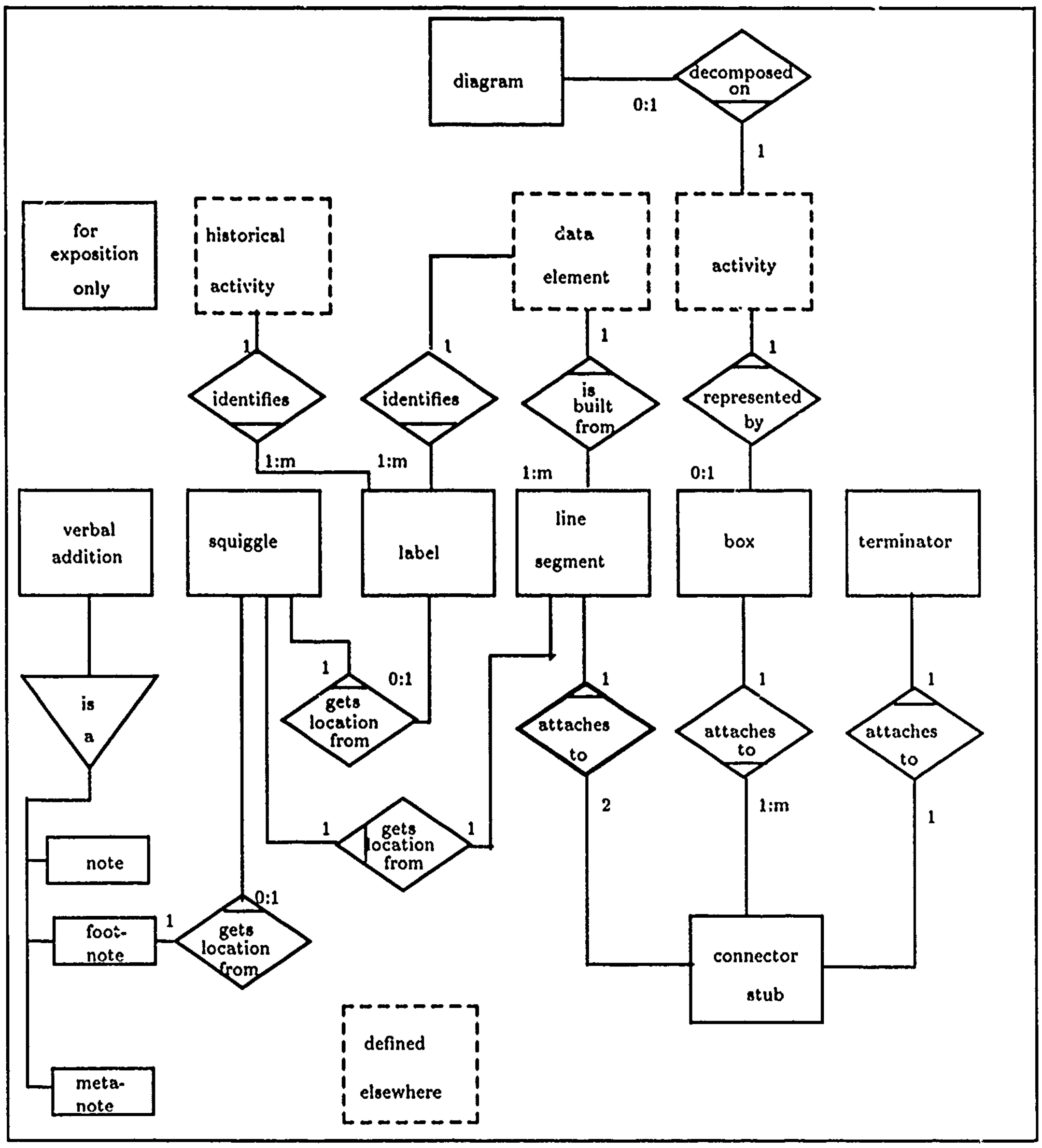


Figure 3.9. IDEF$_0$ Drawing Data Model (Entities and Relationships)

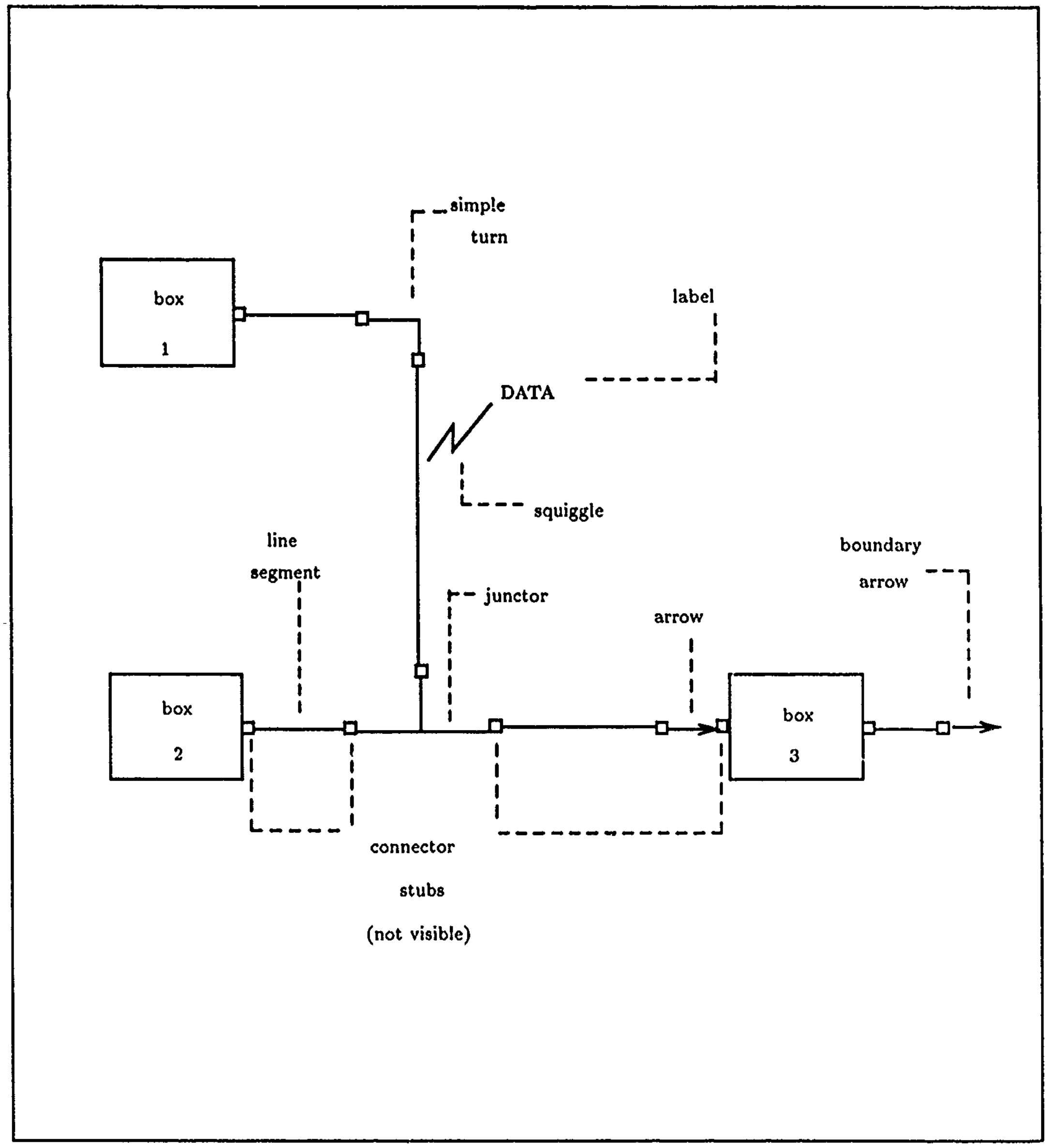


Figure 3.10. IDEF$_0$ Drawing Data Model Illustration

application. The MAGSE should sit on top of a window system, shielding the application from it. Why does Ada need a MAGSE like this? One reason is to contribute to the rather small amount of window and graphical support research written for Ada. A second reason is the intricacies and numerous subtle changes needed in an application when converting it from one kind of window system to another. A third reason is the amount of detail involved in creating a window with its many attributes and then performing event-checking on those windows. A fourth reason is the reliance of an application on a specific window system to supply sophisticated windows such as menus or dialog boxes. What makes the MAGSE machine-independent? The MAGSE is designed to be machine-independent in as far as Ada and the underlying window system are independent.

3.5.2 The Object-Oriented Design Using the requirements analysis for the MAGSE, and taking an object-oriented design approach, this reseach effort divided the MAGSE into seven classes. The objects for these classes are shown in Figure 3.11. Each class contains one or more objects plus methods (operations) for constructing or modifying those objects (4:75–83). The classes are.

- drawing primitive
- 2-D plane
- 2-D matrix stack
- 3-D pyramid
- 3-D matrix stack
- input device
- window manager

The drawing primitive class contains lines, rectangles, circles, and text strings. It also has methods to draw and erase each of these primitives (9:25). One or more drawing primitives can be used to construct complex drawing objects.

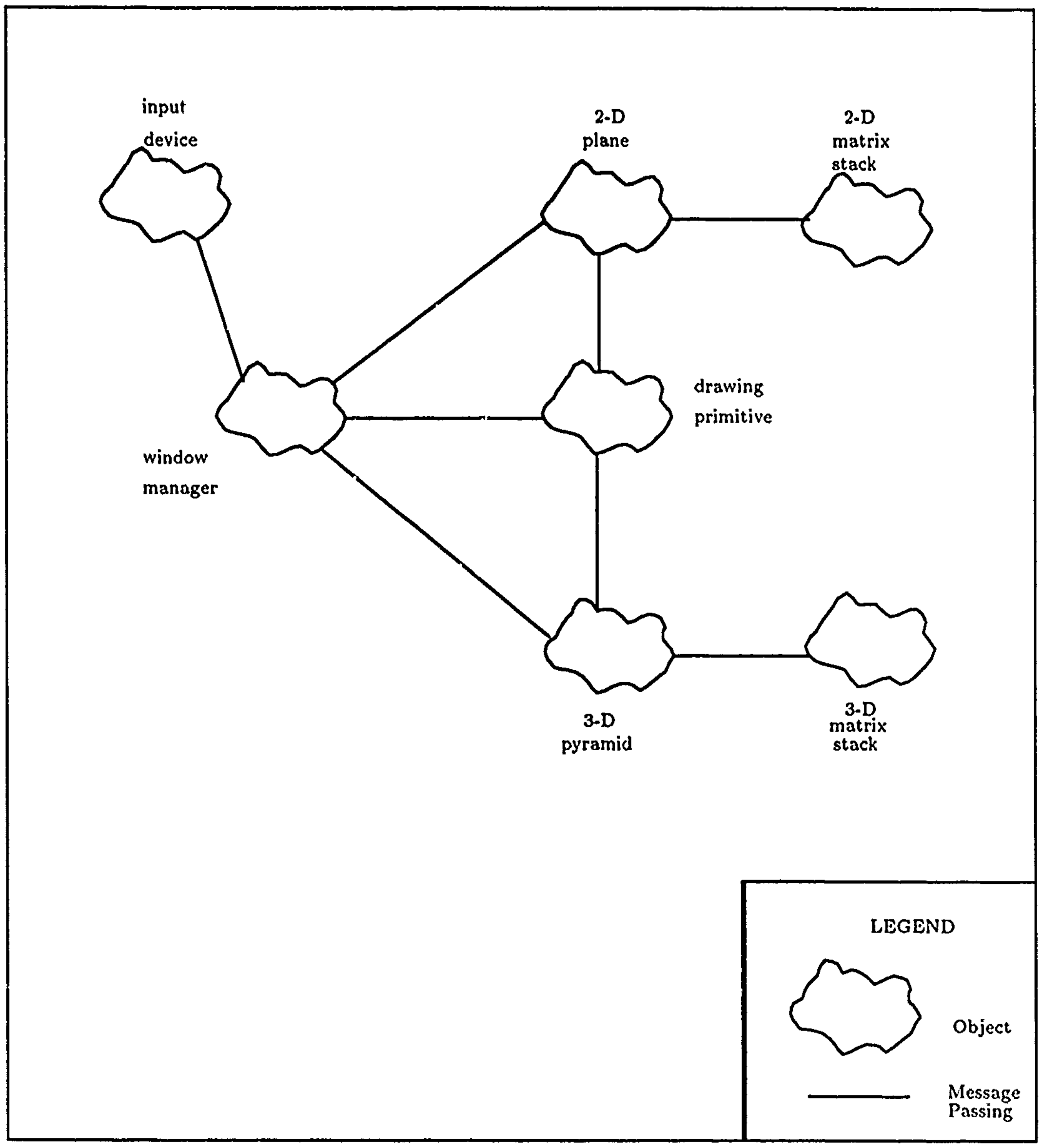


Figure 3.11. Machine-independent Ada Graphical Support Environment Design

The 2-D plane class contains a two-dimensional plane. It also has methods to set the plane's X and Y dimensions, and clip and render complex primitives in the plane (9:67).

The 2-D matrix stack class contains a stack for storing matrices which are used in performing two-dimensional transformations. It also has methods to push a matrix on the stack, pop a matrix off the stack, multiply another matrix times the matrix on the top of the stack, and perform two-dimensicnal rotate, scale, and translate operations on the top matrix of the stack (9:201).

The 3-D pyramid class contains a three-dimensional perspective pyramid. It also has methods to set the X, Y, and Z dimensions of the pyramid, set the viewing location and viewing perspective of the pyramid, and clip and render complex primitives in the pyramid (9:229).

The 3-D matrix stack class contains a stack for storing matrices which are used in performing three-dimensional transformations. It also contains methods to push a matrix on the stack, pop a matrix off the stack, multiply another matrix times the matrix on the top of the stack, and perform three-dimensional rotate, scale, and translate operations on the top matrix of the stack (9:213).

The input device class contains a keyboard, a cursor and a 3-button mouse. It has methods to read the keyboard input, get the cursor position, detect mouse movement, and detect which mouse button was clicked (9:347) (25). It also has methods to detect when a window event occurs with these windows (17:351).

The window manager class contains a window manager object. This object has methods to allocate and deallocate window storage and to retrieve a specific window from storage (9:439). It allocates drawing windows, acknowledge windows, confirm windows, dialog windows, column menu windows, sign windows, and text windows (9:358) (17:351). The class has additional methods to create, display, hide, and destroy these windows (25).

3.5.3 Separation of Graphical Support Environment and Application The design of the MAGSE, as shown in Figure 3.11, is based on having the graphical support environment exist completely separate from a specific application above it and a specific window system below it (2). This prevents the need of any vendor-specific window system calls to be made from anywhere in an application. The description in Figure 3.11 served as the design specification for the MAGSE.

The MAGSE design fills the gap for an easy-to-use window and graphical environment that is portable, written in Ada, and therefore relatively independent of any specific window system application. The catalyst for the creation of the MAGSE was the window and graphical requirements of SAtoolII, but the MAGSE design allows other Ada applications to take advantage of the products and services it offers . Refer to Chapter 4 and Appendix B and C for more information on the MAGSE.

3.6 Graphical User Interface

3.6.1 Requirements Analysis for the SAtoolII Graphical User Interface What functions are needed in the graphical user interface for SAtoolII? Should it have everything the user wants plus anything the designer thinks should be there? To answer these questions this research effort first involved reviewing the information on user-suggested changes to SAtool and the screen format that Johnson had used (see the section on SAtool). Then, it involved pulling ideas from articles by Barth (2), Myers (24), Sabella (28), and Grudin (11). Next, it involved reviewing the graphical user interfaces and menu items of a number of interactive programs including CASE tools, word processing programs, paint programs, desktop publishing programs, and circuit design software. The material mainly looked for in these programs were the operations that the computer and display could do that the user did much slower or could not do at all. One example is a thumbnail view of all diagrams in a project. Another is automatic rerouting of lines. A third is the ability to store, search for, and recall any diagram by name only.

3.6.2 The Object Oriented Design After reviewing numerous graphical user interfaces and collecting interface ideas, I started the design of the SAtoolII graphical user interface by identifying the objects in the user interface. These objects and their window contents are described below.

1. The main screen window objects for the user interface design are:

   - Title window. This window identifies the software and its purpose.
   - Help window. This window displays a message describing the purpose and use of any highlighted menu or sign.
   - Main menu window. This window contains the main menu button objects.
   - Drawing window. This window is where the user creates and modifies an $IDEF_0$ diagram. It is also where various views of a project appear.
   - Drawing Objects window. This window contains the drawing objects used in an $IDEF_0$ diagram.
   - Tools Window. This window supplies the user with the methods used to set flags for SAtoolII to know which tool function to use to create or modify a selected drawing object.

2. The drawing objects for the user inteface are box, FEO, footnote, label, line segment, metanote, note, squiggle, terminator, and connector. These objects are taken directly from the drawing model design.

3. The main menu button objects for the user interface are:

   - Project menu - This main menu button supplies a list of methods used on a whole $IDEF_0$ project.
   - Diagram menu - This main menu button supplies a list of methods used on an $IDEF_0$ diagram.

- Dictionary menu - This main menu button supplies a list of methods used on activity and data element entries in a data dictionary.
- Taxi menu - This main menu button supplies a list of methods use to find a diagram if it exists in the $IDEF_0$ project and move it into the drawing window.
- View menu - This main menu button supplies a list of methods used to create different views of the diagrams in an $IDEF_C$ project
- Options menu - This main menu button supplies a list of methods used to change user-defineable options referenced by SAtoolII to determine the user interface appearance and degree of service.
- Other menu - This main menu button supplies a list of other miscellaneous methods offered by SAtoolII.

After identifying the objects, this research effort involved designing the specifics of the graphical user interface by putting together a user's manual. This user's manual served as the design specification for the graphical user interface component of SAtoolII. Refer to the SAtoolII User's Manual in Appendix E for more information on the graphical user interface. The user's manual goes into more detail on what each of the window objects, drawing objects, and main menu objects are used for.

3.6.3 Intended Use and Expected User for SAtoolII The menu selections in the main menu are designed to serve the user in putting together a project using $IDEF_0$ rather than telling him or her what to do to put a project together. In other words, the user should already know what $IDEF_0$ can do...SAtoolII should make it possible to do it faster and better.

3.6.4 Help from a Graphical Support Environment The design of the graphical user interface for SAtoolII assumes that a separate graphical support environment will supply:

- Drawing, acknowledge, confirm, dialog, column menu, sign, and text windows
- Complex object transformation and rendering in two-dimensions and three-dimensions
- Cursor, keyboard, and mouse input handling
- Multiple window management
- Graphical IDEF$_0$ symbols

The MAGSE design provides all these products and services except for the graphical IDEF$_0$ symbols. These symbols are in the design of the IDEF$_0$ complex drawing class.

3.7 IDEF$_0$ Complex Drawing Objects

3.7.1 Complex Objects from Primitive Ones The IDEF$_0$ language mainly consists of special graphical symbols. The SAtoolII design takes the description of these symbols and indicates how to draw them. Specifically this is done by the IDEF$_0$ complex drawing class. In this class are the following IDEF$_0$ symbols as identified by the IDEF$_0$ drawing model: diagram, box, FEO, footnote, label, line segment, metanote, note, squiggle, terminator, and connector. The MAGSE design supplies the drawing primitives that can be combined to create these IDEF$_0$ complex drawing objects. The drawing primitives are a line, rectangle, circle, and text string. The MAGSE design also provides the ability to graphically scale, rotate, and translate these complex drawing objects for two-dimensional or three-dimensional rendering. The IDEF$_0$ manual (23) and the revised IDEF$_0$ drawing model served as the design specification for the complex drawing class component of SAtoolII.

3.8 Project Facts and CLIPS

3.8.1 Format for the Project Files In order to expand the expert system use of the files produced by SAtoolII, this research effort cooperated with Kitchen (20) in designating a file format for the project files read and written by SAtoolII. The format we chose was a CLIPS fact file format that also qualifies as an expert fact file format. CLIPS stands for the C Language Integrated Production System. Here is an sample fact statement:

```
(deffacts box "box object"
   (has_name      Activity_1)
   (has_location  X          4)
   (has_location  Y          8) )
```

The use of facts, or predicates, to save the information about an entity-relationship model follows naturally from the purpose of predicates. According to Dromey, "A predicate is a formula that may be used either to ascribe a property to an entity or to assert that certain entities stand in some relationship." (8:57)

In other words, predicates can serve to identify the attributes of an entity and also specify relations that hold between two or more entities.

3.8.2 IDEF$_0$ Fact and File Classes Using this fact format as a basis, the fact and file objects for SAtoolII were identified and inter-related as illustrated in Figure 3.12. The Essential Fact Utilities and the Drawing Fact Utilities conform to what Booch defines as class utilities containing free subprograms (4:82,160). These free subprograms have visibility to two or more objects and permit access to all the essential model objects and all the drawing model objects. The essential and drawing model information passes between the fact utilities and the file objects through the use of a buffer. In addition to the file objects is a CLIPS working memory object. This working memory object complements the expert system aspect of SAtoolII. The class and utilities descriptions in Figure 3.12 served as the design specification for the project facts input and output component of SAtoolII.

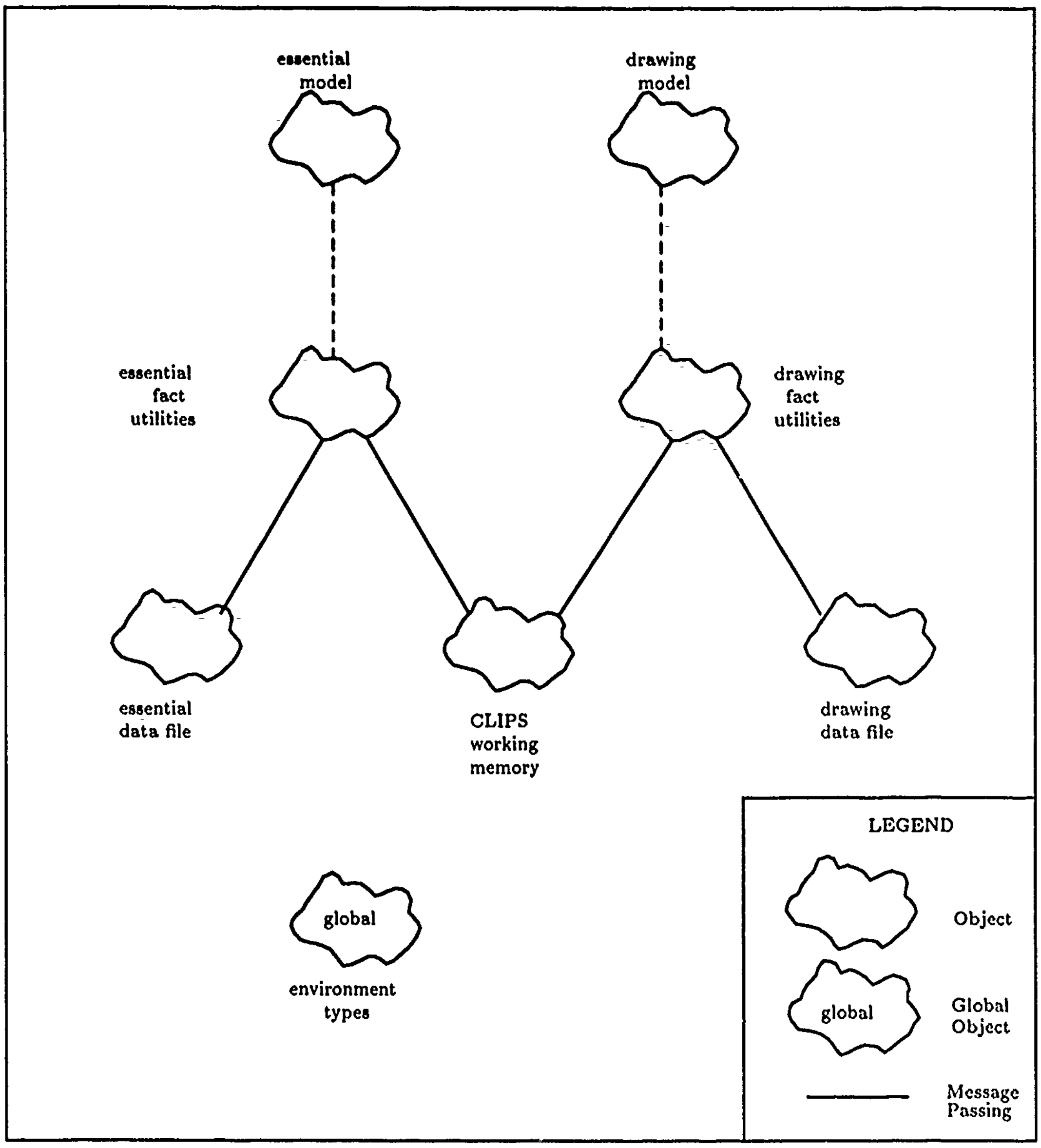


Figure 3.12. SAtoolII Project Fact Utilities and Files Design

3.8.3 Format for Miscellaneous Files Along with the project fact files, SAtoolII will also produce ASCII files containing facing page text, data dictionary information, error information, and program usage data. These files are for output only.

3.9 Error Handling

In an effort to centralize the error handling and error messages in SAtoolII, the SAtoolII design contains an error handler class with an error handler object. This design calls for an error indicator to be passed from lower design levels of SAtoolII to the higher design levels until it is trapped in the main subprogram. The error handler object then reacts to inputs submitted to it concerning an error number, an error location, and a flag indicating that the error that occurred is known or unknown. The error handler then informs the user of the error through the use of an acknowledge window. It also uses a confirm window to ask the user if he wishes to continue program operation or let the error continue on and abort the program. This error handling description served as the design specification for the error handler component of SAtoolII.

3.10 Macro Operations and Model Constraint Management

The entity-relationship approach to object-oriented design does a good job of identifying entities and their attributes and relationships. This approach encapsulates an entity with attribute and relationship methods that keep its state and operation autonomous from any other entity. Because of this autonomy factor however, another component is needed to contain inter-model and intra-model macro operations and project integrity constraint management methods. From a bottom-up view, these methods provide the upper-level link and functionality for the essential model, drawing model, complex drawing objects, and facts and expert system utilities. From a top-down view, these methods provide the lower-level link and functionality between the graphical user interface and these components.

3.11 SAtoolII - An Integrated Design

The design of SAtoolII consists of many components. By designing SAtoolII in this manner each component can be individually implemented, tested, and then integrated together. Figure 3.13 illustrates this integration. The components of the SAtoolII design are:

- Machine-independent Ada Graphical Support Environment (MAGSE) component
- $IDEF_0$ essential model component
- $IDEF_0$ drawing model component
- graphical user interface component
- $IDEF_0$ complex drawing objects component
- file and expert system utilities component
- error handler component
- inter-model and intra-model macro operations and project integrity constraint management component

The description in Figure 3.13 served as the design specification for the component integration of SAtoolII.

3.12 Summary

This chapter has taken the reader through the design phase of SAtoolII. The chapter summarized Johnson's work on SAtool in C and gave the background for the essential and data model designs and the revisions made to them. It then explained the design of the machine-independent Ada graphical support environment called MAGSE, the graphical user interface, the complex $IDEF_0$ drawing objects, the project file format, and the centralized error handling for SAtoolII. It also pointed out the needed for explicit macro operations and project integrity constraint management

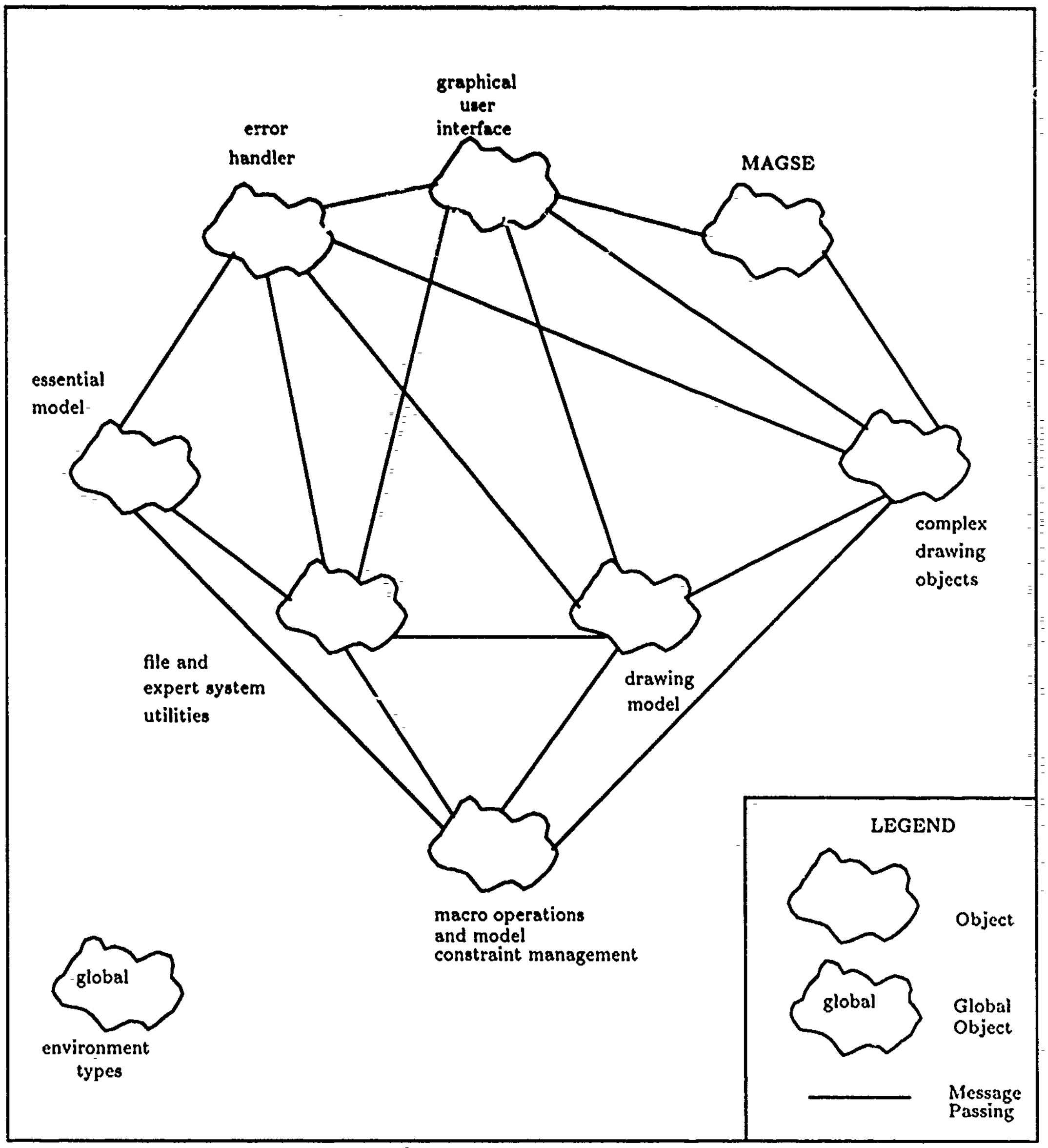


Figure 3.13. SAtoolII Integrated Component Design

methods to form the link between the basic components of SAtoolII and the graphical user interface. At the end, the chapter pulled all design components together into an integrated design specification for SAtoolII.

IV. SAtoolII Implementation

4.1 Introduction

This chapter takes the reader through the incremental implementation, in the Ada programming language, of the components specified in the design of SAtoolII. The next section begins by describing the generic multiple object manager that is used to keep track of the essential model and drawing model objects. The third section explains how the essential and drawing models were implemented by applying ideas from Shlaer and Mellor (30) and Kiem (19) to the model design. The four and fifth sections cover how the machine-independent Ada graphical support environment and the graphical user interface were implemented. The sixth and seventh sections describe the implementation of the IDEF$_0$ complex drawing class and the project facts utilities and files. The eighth section deals with the error and exception handling. The ninth section describes the partial implementation of the macro operations and project integrity constraint management routines. The tenth section covers the incremental steps used in partially implementing SAtoolII.

Each of the packages in SAtoolII *withs* in the Environment Types package. This package contains constants, types, exceptions, and utility functions needed by each of the packages.

4.2 Generic Multiple Object Manager

4.2.1 The Need for a Multiple Object Manager The design of the essential model and drawing model brought the need of an implementation package that created, modified, and kept track of a number of objects of a specific type. This multiple object manager was visualized as a linked list with functions and procedures to perform these functions. Because each list of objects had a common subset of functions, the multiple object manager looked like a good candidate for a generic package. After making a list of features that each for the manager, Kitchen took on the task of implementing the generic multiple object manager package. He based his implementation mainly on the Booch component known as a Queue Nonpriority Balking Sequential Unbounded Unmanaged Iterator (3:71-96).

4.2.2 Modifications to the Nonpriority Balking Queue Component In implementing the generic multiple object manager, Kitchen made some modifications to the ideas presented by Booch. He listed them briefly (20):

- The passive iterator was replaced with an active iterator.
- Since the iterator is now active, functions and procedures can accept the iterator as a parameter instead of a position number.
- Because an active iterator now exists and thus a pointer into the manager is available, the function Position_Of, which returned a natural number, was not necessary and was removed.
- The procedure Set_Item was added to permit an item to be updated in place. This cuts down system garbage collection and saves time when an item has multiple attributes to be updated or added.
- The procedure Add_Item was modified to include the iterator as a parameter that would be left pointing to the just-added item.
- The procedures Copy, Pop, and Front_Of were determined unnecessary and were removed, but they can easily be added back again if warranted.

4.2.3 The Generic Multiple Object Manager Package The generic multiple object manager package accepts a record type parameter. It then supplies a manager type and subprograms to create, modify, and keep track of objects with the designated record type. The key field in each object record is the object name.

4.3 Essential and Drawing Models

4.3.1 The Need for a Transformation Methodology The information on the SAtoolII design in chapter III covered some of the work done by Smith and others in designing an entity-relationship model for IDEF$_0$. After Kitchen had revised the essential model and this research effort the drawing model part of the IDEF$_0$ model, areas of cooperation were identified in implementing the two models. One area was the use of a multiple object manager, described in the section above. A second area was in the actual transformation of the entities, attributes and relationships into an Ada implementation. Ideas were used from Shlaer and Mellor (30) on object-oriented data modeling, from Kiem (19) on the Keystone Methodology, and from this research experience in redesigning the essential and drawing models (20).

4.3.2 Transforming an Entity-Relationship Model into Ada Using the knowledge gained and lessons learned in implementing the essential model and drawing model, this research effort formulated the following transformation steps. These steps describe the methodology used in transforming the IDEF$_0$ essential model and drawing model into actual Ada source code.

4.3.2.1 Step One: Create an E-R Model Create an entity-relationship model of a system to identify the entities, attributes, and relationships in the system. Label all relationships as one-to-one, one-to-many, or many-to-many.

4.3.2.2 Step Two: View the Entities as Objects Look at the entities as objects and consider the implementation of the system using these objects. Determine if these objects completely identify all the objects of the system being modeled.

4.3.2.3 Step Three: View the Many-to-Many Relationships as Objects Look at the many-to-many relationships as objects and consider the implementation of them as correlation table objects. For the many-to-one relationships consider putting a referencing attribute field in the *1* object. For the one-to-one relationships, consider to which object to give the referencing attribute field.

4.3.2.4 Step Four: Modify the E-R Model Modify the entity-relationship model to reflect the lessons learned from steps two and three. Continue iterating through steps two and three until the entities, attribut[illegible] nd relationships completely model the system. To test the system, use *what if* situations that the system should be able to handle, and walk through the model to check if the entities, attributes, and relationships *model* the particular situation.

4.3.2.5 Step Five: Code the package arrangement for the E-R Model Code each entity and many-to-many relationship of the model as an object. Code the descriptive attributes of relationships or entities as record fields in each object. Include in the many-to-many relationship data structures a relationship tuple containing referencing attributes. Assign each object a type class package, an object manager package, and an input/output package. In the manager and input/output packages, *with* in the types class package.

4.3.2.6 Step Six: Code the operations for the objects in the E-R Model In the object manager package create constructor procedures to set descriptive attributes and referencing attributes for entities on the *one* side of a one-to-many or one-to-one relationship. Create selector functions to retrieve attributes and all relationships. The constructors and selectors should completely represent all the visible routines from an object manager package. The types utilized in the parameters for these routines should completely represent all visible types from an object manager package. This completeness and visibility will be correct if the entity-relationship model completely contains all entity attributes and entity relationships.

If any additional *visible* constructors or selectors need to be added to an object manager package, examine the entity-relationship model to see if a corresponding attribute or relationship exists for the new routine. If one does not exist, one of the following is probably true:

- The routine is a generalization of another currently visible routine
- The routine corresponds to an attribute or an entity relationship that was previously overlooked
- The routine has been incorrectly made visible and should only appear in the body of the object manager package

4.3.3 The Multiple Object Manager and Inheritance Kitchen and this research effort differed in the model implementations in the number of multiple object manager types that were instantiated. Because of the variety of attributes that each entity and relationship had in the essential model, Kitchen instantiated a generic multiple manager type for each entity and many-to-many relationship (20). This research effort chose to instantiate just a single object manager type, called a drawable object manager type, to handle all the graphical objects (or entities) in the drawing model. This single type was made possible by using a kind of attribute inheritance with the graphical entities.

To use the inheritance concept, this research effort set up a super class object type called a drawable type and subclass types for each of the graphical objects. The first part of the drawable type contains the attributes common to all drawables, such as name and location. The second part contains both a discriminant unique to each drawable subclass and attribute fields to accompany the discriminant. By using the super class and subclass arrangement, complex objects can be built by linking together subclass objects as described in Barth (2). The complex object can then be manipulated as if it were a single drawable. An example of a complex object is a diagram. When nothing is attached to a diagram it serves as a simple graphical object; but multiple objects can be placed on it. After this relationship is established, whatever happens to the diagram also happens to all the objects placed on it. This includes such actions as storage, display, moving, scaling, and deletion.

4.3.4 The Ada Packages for the Essential and Drawing Models Figure 4.1 and Figure 4.2 show the Ada package implementation of the essential and drawing models. Each of the packages has no procedure or function coupling with the other packages. All ties among the various drawing objects are maintained through the use of drawing object name strings. Because the diagram object maintains a list of the drawable objects that are on a diagram, it *withs* in the packages in order to use their operations. Refer to Appendix F for more information on the actual source code.

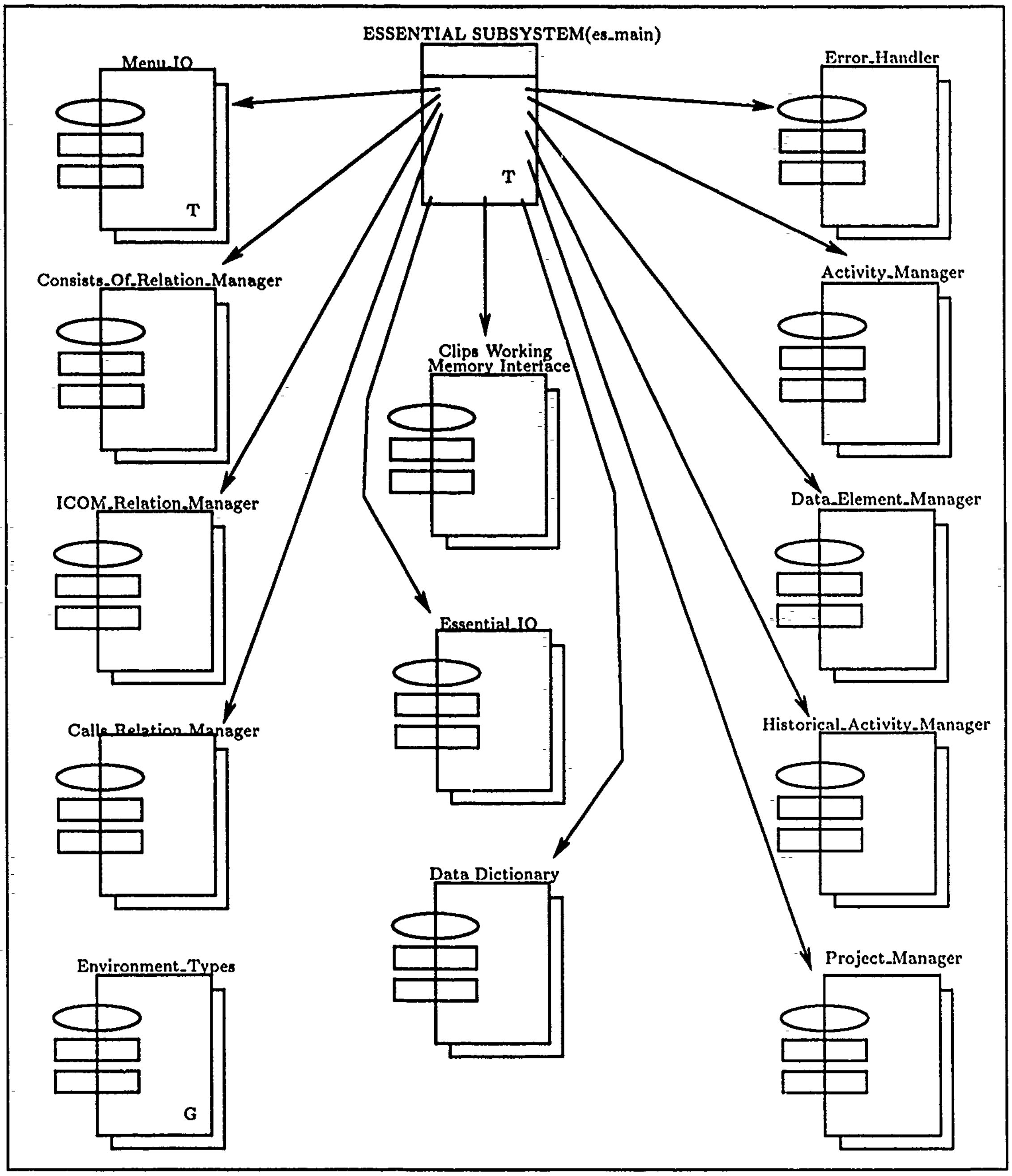


Figure 4.1. IDEF$_0$ Essential Data Model Implementation Packages

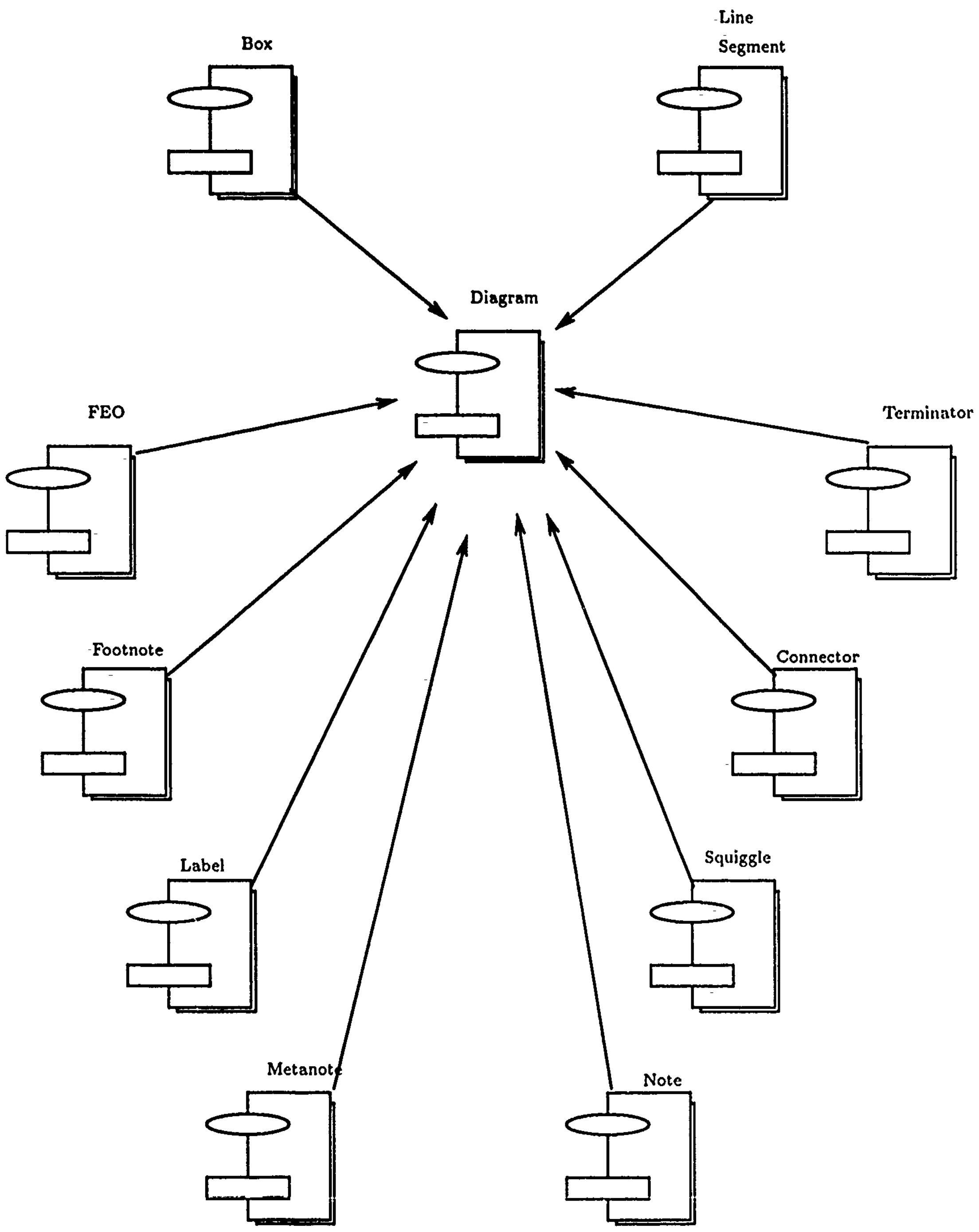


Figure 4.2. IDEF$_0$ Drawing Data Model Implementation Packages

4.4 Machine-Independent Ada Graphical Support Environment (MAGSE)

4.4.1 Comparison of Design Goals with Implementation Factors The machine-independent Ada graphical support environment or MAGSE was designed to provide an interface between any window system and an Ada application. It was also designed to sit on top of a window system, shielding the application from it and thereby creating a well-defined line of separation between the application and its graphical needs. These design considerations were one stimulus that drove the implementation of the MAGSE. The other two were the complicated nature of the X Window System and the SAIC Ada interface to it.

4.4.2 The Breadth and Depth of the X Window System The Massachusetts Institute of Technology (MIT) designed the X Window System with window mechanisms rather than window policy (25). Because of this intention, the X Window System can be used to mimic another window system while taking advantage of the networking capabilities of X. This is this generic window ability brings complications and complexity with it. To just create a window, establish its many graphical properties, and customize it to a certain style takes at least 24 X library function calls. In addition, ten of those calls return values or data structures that are needed as parameters to subsequent function calls that involve the window. After analyzing the source code of numerous applications, including a two-dimensional graphics tool (5), this research effort decided to capture all the complicated structures into a single record type and the numerous function calls into a few macro-functions. The result is the machine-independent Ada graphical support environment called the MAGSE. Some window flexibility is lost using the MAGSE, but much easier and simpler window handling is gained.

4.4.3 The SAIC Ada Bindings to the X Window System The object code for the window and graphics functions available in the X Window System are contained in a library called xlib. The functions in xlib are written in the C language format, which means calls to the functions and the parameters must conform to the way C does business (25). Ada provides a compiler directive called a pragma interface which allows calls to C libraries to be made from Ada programs. Appendix A goes into more detail on how these pragmas work and the pragmas and data structures that SAIC developed to enable an Ada program to make xlib calls (15). Through the use of the SAIC Ada interface (Ada bindings) to the X Window System's xlib, an Ada program can become an X Window client application.

4.4.4 The MAGSE : A Level of Abstraction In developing the Ada bindings to the X Window System, SAIC for the most part followed the naming conventions established by MIT for the X Window functions, error flags, event masks, and symbols (constants). However, they also applied the packaging and strong typing capabilities of Ada to the function groupings and the function parameters. The MAGSE provides a level of abstraction between an application program and the complexity of the xlib and the Ada bindings to the xlib . The MAGSE also provides specialized windows, such as menus and dialog boxes, that are not a part of the X Window System xlib. The aspiring Ada programmer doesn't have to spend days studying an xlib reference manual, examining sample X Window programs in C and Ada, and searching through the SAIC source code for parameter types and package names. He or she can just *with* the MAGSE packages into the application, and then declare variables and make subprogram calls as directed by the MAGSE package specifications. By doing this, the application will become a genuine X client.

4.4.5 The MAGSE Implementation The MAGSE implementation consists of the Ada packages shown in Figure 4.3. The MAGSE globals package contains all variables, constants, and types shared among the other MAGSE packages. The drawing primitive package contains routines to create lines, rectangles, text, and circles. The input device package contains routines to gather information from the mouse and keyboard and to utilize pull-down menus, confirm boxes, acknowledgement boxes, dialog boxes, and signs (buttons) for entering information into a program.

The MAGSE contains two packages to aid in two-dimensional drawing and two packages to aid in three-dimensional drawing. These packages correspond to four of the classes in the MAGSE design: Plane2D_Class, Matrix2D_Class, Pyramid3D_Class, and Matrix3D_Class. These packages were implemented using the two-dimensional and three-dimensional graphics principles explained in the text by Foley and van Dam (9:201) (9:229). The two-dimensional packages provide a two-dimensional plane and a matrix stack. The plane can be stretched over a MAGSE window and used to produce two-dimensional graphical images that can be scaled, translated, and rotated. The three-dimensional packages provide a three-dimensional pyramid and a matrix stack. The X and Y axes at the base of the pyramid can be stretched over a MAGSE window so that the pyramid can be used to produce three-dimensional graphical images that can be scaled, translated, and rotated (9:280). Refer to Appendix B for more information on the services offered by these four packages.

An application's view of the MAGSE is the MAGSE interface package. This package contains the package specifications for all the MAGSE package bodies. The MAGSE interface package contains no sign of complicated structures or numerous function calls to the X Window System or any other window system. All this detail is kept secure in the various package bodies. The application sees only a window identification type, choices of specialized windows, basic window manipulation routines, and basic drawing primitive routines. In the MAGSE package bodies, the internal structures, functions, and procedures are tailored to the specific window system that the MAGSE will be placed on top of. By implementing the MAGSE package bodies first for the X Window System, its structures and routines dictate mechanisms but no policy. The only restrictions on the window system are those imposed by the speciali windows offered by the MAGSE to an application. These windows (drawing, acknowledge, confirm, dialog, column menu, sign, and text) are not provided by the X Library (xlib) of the X Window System, but instead are implemented right in the MAGSE. Some of the ideas used by this research effort in implementing these specialized windows came from the X Window two-dimensional drawing tool developed by Cheng (5).

SAtoolII with its graphical user interface is just one example of an Ada application that

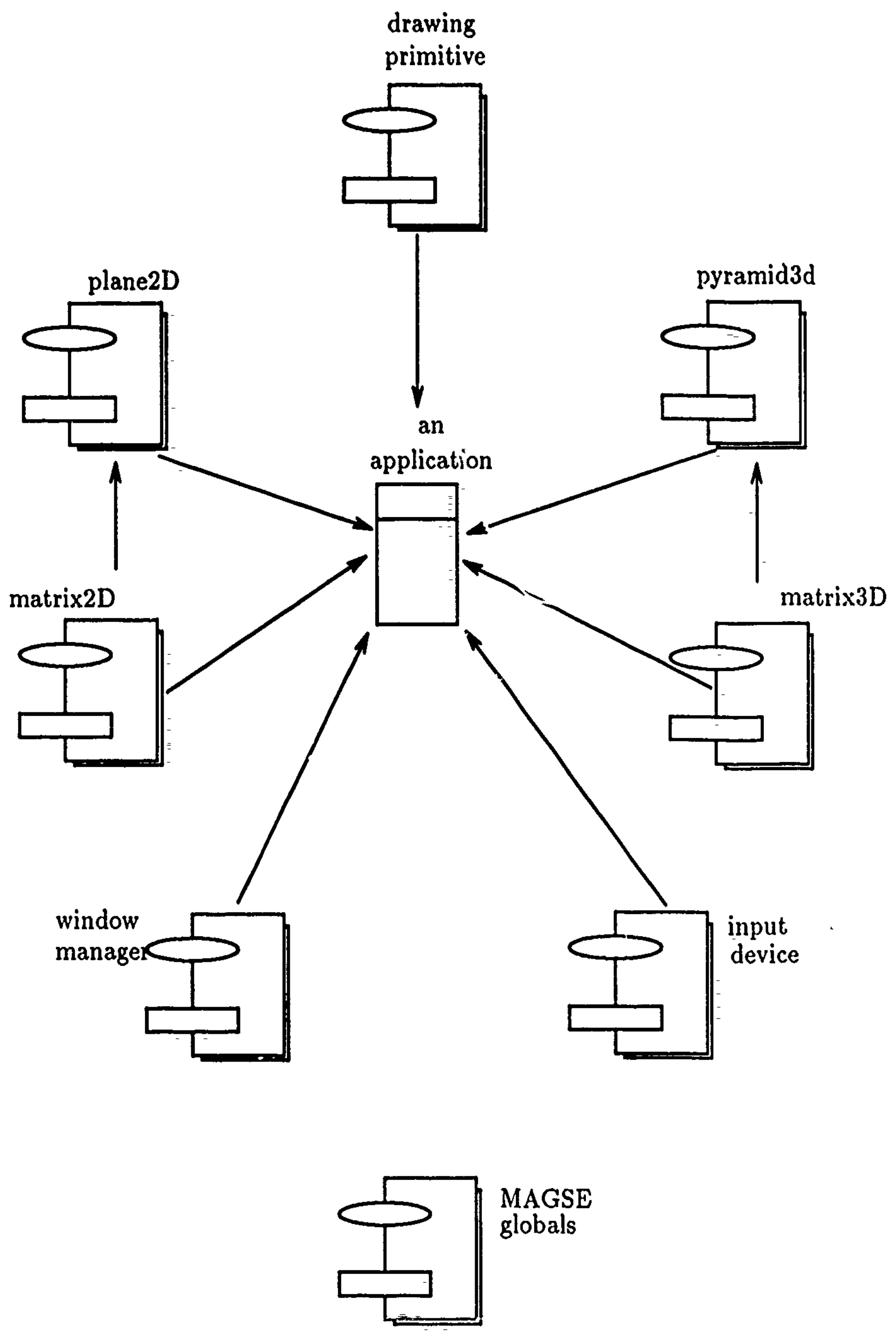


Figure 4.3. MAGSE Implementation Packages

takes advantage of the MAGSE. Refer to the MAGSE Reference Manual in Appendix B for more information on the products and services the MAGSE has to offer.

4.5 Graphical User Interface

4.5.1 Packages for the Graphical User Interface The main screen window objects making up the SAtoolII graphical user interface (GUI) design dictated the names and purposes for the highest level Ada packaging for the interface. These packages are called Title, Help, Main Menu, Drawing, Objects, and Tools. Inside the main menu package are package declarations for each of the main menu button objects. Inside the objects package are package declarations for each of the drawing objects buttons. Figure 4.4 shows how all these packages fit together. This packaging scheme keeps in step with the object-oriented design approach to SAtoolII versus the functional approach taken by the menu level of SAtool (See the discussion in Chapter III of this thesis on SAtool in C).

4.5.2 What The Graphical User Interface Does and Doesn't Do Above the level of the graphical user interface packages in SAtoolII is the main subprogram. This main SAtoolII subprogram only acts as a control for the user interface packages; it does not supply the methods (procedures and functions) for the user interface objects. The work of the graphical user interface remains completely enclosed in the window object packages. The methods supplied in the packages of each of the graphical user interface objects are the menu selections described in the SAtoolII User's Manual. Because of time constraints, this research effort only implemented a prototype of the SAtoolII graphical user interface. This prototype contains a variety of MAGSE input devices. These input devices allow a user to list and select all the features described in the SAtoolII User's Manual, but with no actual functionality.

The fully-functional SAtoolII graphical user interface will serve as a liaison between the user and the lower levels of SAtoolII. It will make SAtoolII easier and more pleasant to use. Refer to

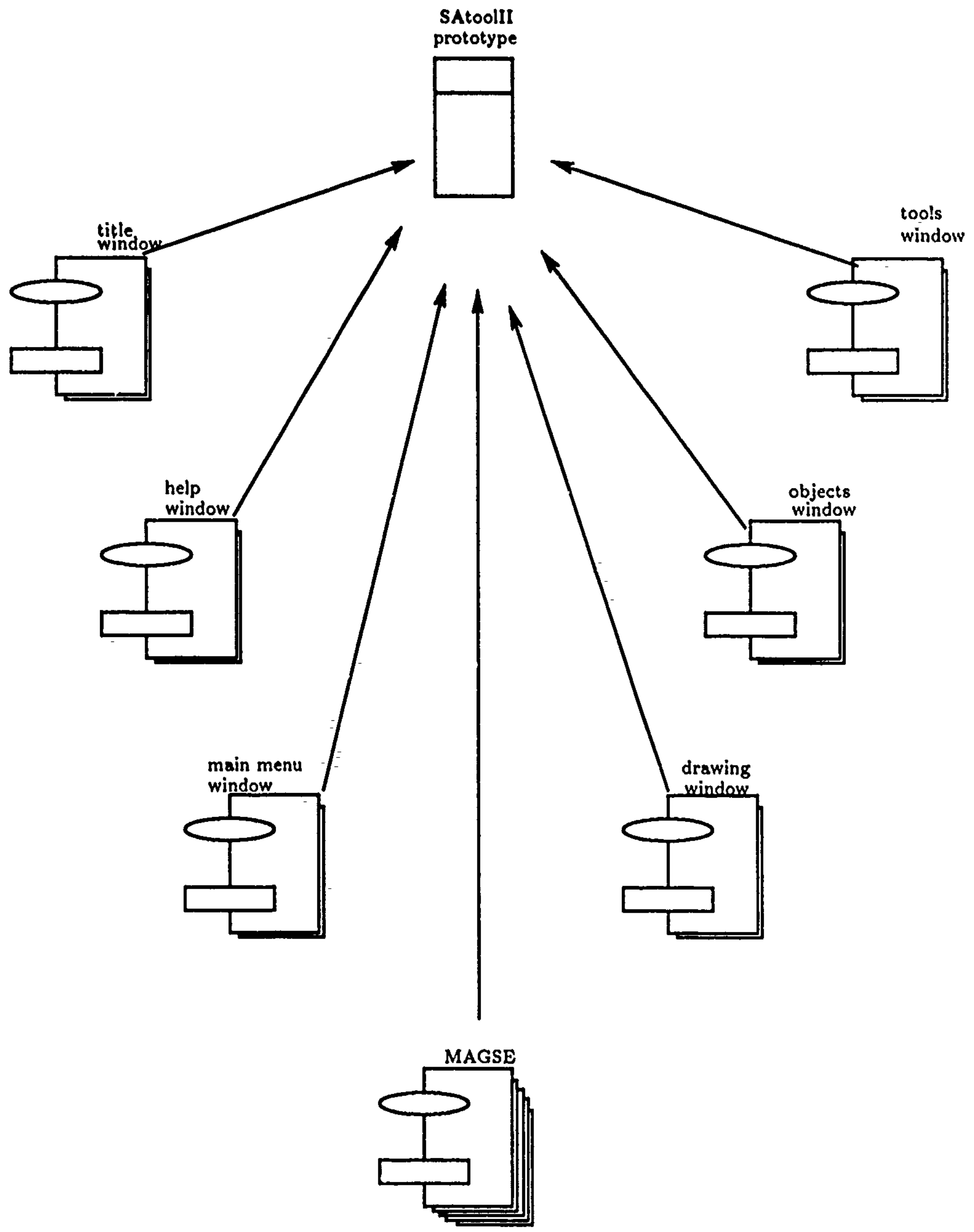


Figure 4.4. SAtoolII Graphical User Interface Implementation P :kages

the SAtoolII User's Manual in Appendix E for specific information on the methods to be made available for each of the main menu button objects and drawing objects offered by the graphical user interface.

4.6 IDEF$_0$ Complex Drawing Objects

4.6.1 IDEF$_0$ Geometry Package The SAtoolII design called for an IDEF$_0$ complex drawing class to contain descriptions of the IDEF$_0$ symbols and methods to draw them. This package was not implemented in this research effort because of time constraints. It should be implemented using the drawing primitive package in combination with either the two-dimensional plane or three dimensional pyramid packages. The use of the plane or pyramid packages will allow a diagram and its drawing object contents to be displayed in any size, angle, or position desired. The drawing primitive package provides the primitive shapes needed to construct the complex symbols used for the IDEF$_0$ notation.

4.7 Project Facts and CLIPS

The project file design for SAtoolII designated that the essential and drawing model information for an IDEF$_0$ project be stored in two files in a CLIPS fact file format. Because of time constraints, this implementation was not done in this research effort. Instead the contents of the drawing model data structures were saved to a text file in the format described in Appendix II.

As a suggested implementation, SAtoolII creates and uses two main project files: an essential model file with a .esm extension, and a drawing model file with a .drm extension. These files contain all the information necessary to return the essential model and drawing model data structures in SAtoolII to the state they were in the moment before the files were saved. The essential model file contains the information on the activities, data elements, and their relationships for a specific project. It also includes some of the non-graphical information for the data dictionary. The

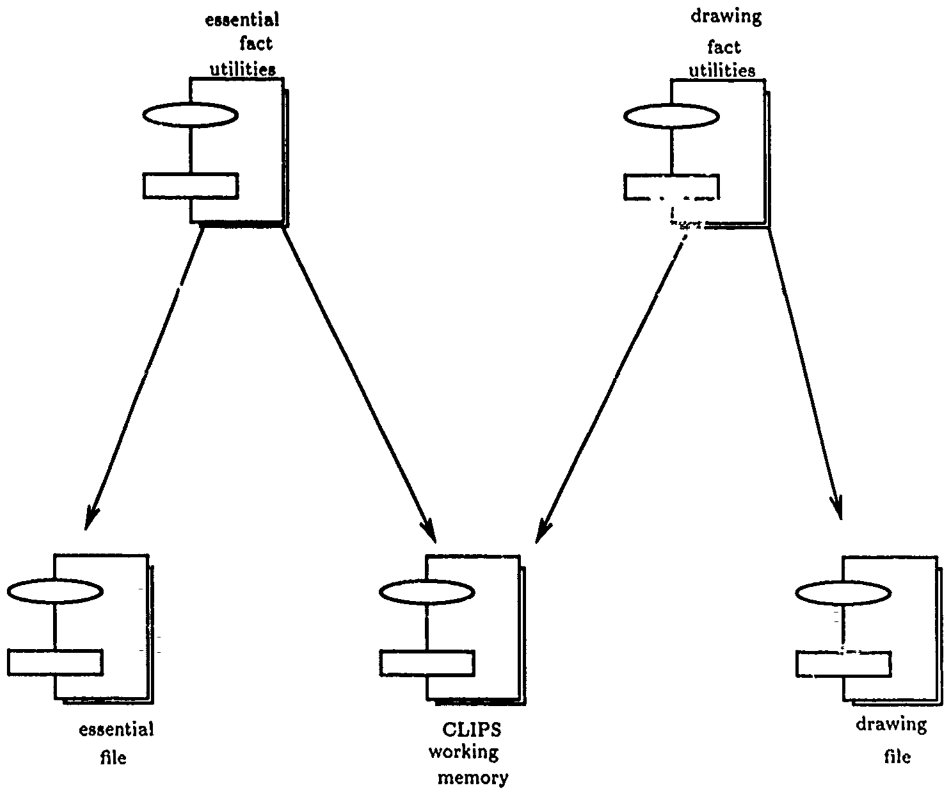


Figure 4.5. SAtoolII Project Fact Files Suggested Implementation

drawing model file contains the information needed to recreate the graphical object attributes and relationships for each diagram in an IDEF$_0$ project. Essential and drawing model fact information is transfered between the internal data structures and the files through the use of two fact utilities packages which use a fact buffer. This suggested implementation is illustrated in Figure 4.5.

Other files are needed strictly for output for SAtoolII. These files are all ASCII files and are used for facing page text, database information, error information, and program usage statistics.

4.8 Error and Exception Handling

The design of SAtoolII calls for centralized error handling. This was not implemented in this research effort because of time constraints. As a suggested approach, this is implemented using the Ada exception feature and by having error location, error number, known exception occurred flag, and unknown exception occurred flag variables in the environment types package. Also in this package are general exceptions for each SAtoolII package. When an error occurs, it is trapped by a subprogram's exception clause. This subprogram (procedure or function) then sets either the known or unknown error flags if neither are set yet and raises the general package error. This exception is then passed up through the package levels by the exception clauses of all calling subprograms until it reaches the SAtoolII main subprogram which calls the exception handler. The exception handler package uses the information in the error number, error location, known exception occurred flag, and unknown exception occurred flag variables in conjunction with the general package error exception to inform the user of the error. This involves the use of an acknowledge window to notify the user and a confirm window asking the user if wants to let the error continue on and abort the program or stop the error at this point and continue program execution.

4.9 Macro Operations and Model Constraint Management

The design of SAtoolII called for packages which contain inter-model and intra-model macro operations and project constraint routines. These packages were only partially implemented in this research effort in the implementation of a drawing model textmode-based driver program. The upper level packages in this driver correspond directly to the SAtoolII graphical user interface packages. In these packages is the source code to add and decompose boxes and source code to build and maintain a project hierarchy of diagrams.

Some of the unimplemented macro operations can be easily coded using the box and hierarchy code as an example. However, because of the autonomy designed into the IDEF$_0$ model, most of the project constraint source code will take some deep thought. Follow-on research should consider how the essential model and drawing model will work together in storing, updating, saving, and loading information (facts) about an IDEF$_0$ project.

4.10 SAtoolII - An Incremental Implementation

4.10.1 The SAtoolII main procedure Above all the packages of SAtoolII is the main subprogram or the main procedure. This procedure is implemented as part of the graphical user interface for SAtoolII. The main procedure initializes the MAGSE, commands the graphical user interface to create its main windows, executes the main event loop for the interface, calls the exception handler when necessary, and terminates the MAGSE.

4.10.2 The Component Integration The implementation of the components of SAtoolII in this research effort have occurred in an incremental fashion. Because of time constraints, certain components were not implemented. The components implemented by this research effort and the work with Kitchen (20) were:

- Essential model
- Drawing model
- Machine-independent Ada graphical support environment (MAGSE)
- SAtoolII graphical user interface
- Part of the file and expert system utilities
- Part of the macro operations and project integrity constraint management

The components left to be implemented are:

- Complex drawing objects
- Error handler
- Remainder of the file and expert system utilities
- Remainder of the macro operations and project integrity constraint management

Figure 4.6 illustrates the overall implementation architecture of SAtoolII after all of the components are implemented and completely integrated. As a means of testing and demonstrating the performance of the implemented SAtoolII components, three programs were developed: the drawing model driver program, the SAtoolII graphical user interface prototype, and the essential model driver program. The first two of these programs were described earlier in this chapter. The essential model driver is explained in the research done by Kitchen (20).

4.10.3 SAtoolII program documentation Appendix D contains the file names, compilation order, and linking instructions to create an executable version of the demonstration programs. Appendix F contains information on the SAtoolII Ada source code. Appendix E is the SAtoolII User's Manual. This manual has sections to match the needs of a variety of users. The sections are:

- Introduction
- Getting Started
- Guided Tour
- Objects and Tools
- Main Screen Menus
- Printing a Window

4.11 Summary

This chapter took the reader through the component implementation, in the Ada programming language, of SAtoolII. The chapter described the packages for the generic multiple object manager, the essential and drawing models, the MAGSE, the graphical user interface, the $IDEF_0$ geometry, the project fact files, the exception handler, and the SAtoolII main procedure. The SAtoolII program was designed to help a user create and modify a project of $IDEF_0$ diagrams, not

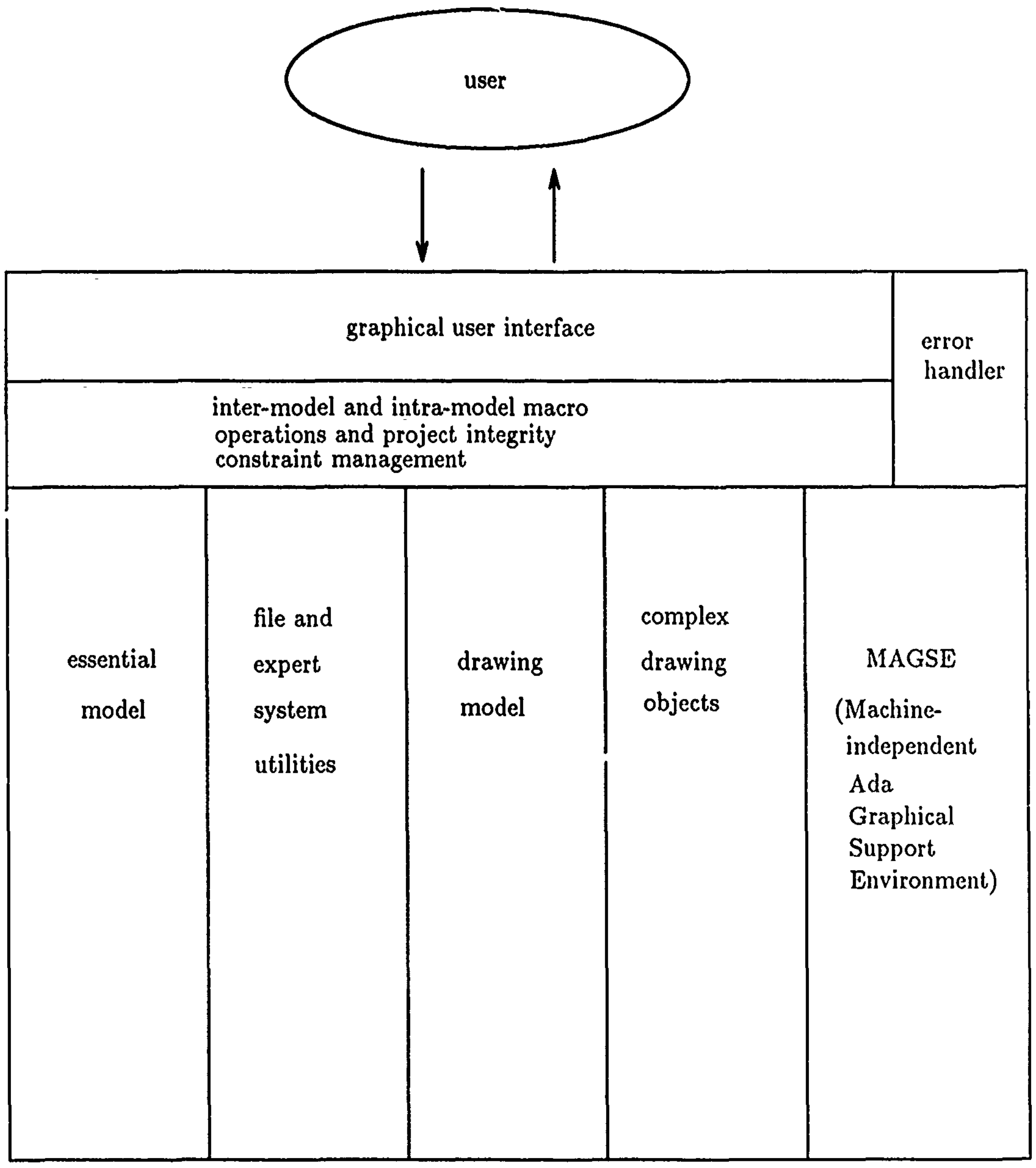

user
graphical user interface
error handler
inter-model and intra-model macro operations and project integrity constraint management
essential model
file and expert system utilities
drawing model
complex drawing objects
MAGSE (Machine-independent Ada Graphical Support Environment)

Figure 4.6. SAtoolII Overall Implementation Architecture

tell him or her how to put the project together. The implementation reflects that design. The user of SAtoolII should find it to be a software package that makes the task of requirements analysis with IDEF$_0$ faster, better, and easier than using just paper and pencil.

V. *SAtoolII Testing and Evaluation*

5.1 Introduction

This chapter takes the reader through the testing and evaluation phase of SAtoolII. The next section describes the different types of tests in the test suite for SAtoolII and the results of those tests. The third section lists the suggested areas that SAtoolII should be evaluated in when the components of SAtoolII are finally all implemented and integrated.

5.2 Testing

5.2.1 Purpose of the Test Suite The test suite for SAtoolII is arranged so that an impartial but knowledgable Ada programmer can perform a variety of tests on SAtoolII. The suite consists of six types of tests: functional, module (package), subsystem, integration, regression, stress, and acceptance testing (32:408-409). The functional, module, subsystem, and stress tests are performed as soon as the packages are coded. This facilitates the incremental implementation of SAtoolII. The integration, regression, and acceptance testing are performed after all the components making up SAtoolII are put together in the SAtoolII program. The major components making up SAtoolII and involved in the subsystem tests are the MAGSE, drawing model, essential model, fact utilities, fact files, error handling, complex drawing objects, constraint management, and graphical user interface.

5.2.2 Test Suite For SAtoolII The directions for performing six types of tests on SAtoolII are given below. All test failures should be repeatable and completely documented.

5.2.2.1 Functional Testing For functional testing, test each procedure and function in each package to ensure they perform the task they were designed to do. Perform these tests by building a driver program for each package in SAtoolII.

5.2.2.2 Module Testing For module testing, modify the driver programs used in the functional testing to provide simple menus so that the procedures and functions in the modules can be executed in a nondeterministic order. Check the modules for both proper operations and for the handling of error conditions.

5.2.2.3 Subsystem Testing Perform subsystem testing on the MAGSE, the essential model, the drawing model, and the graphical user interface. Test the operations of each of these subsystems by using a derivation of the menu-driven driver programs developed for module testing. Execute the major tasks offered by the subsystems and check for both proper operation and error handling.

5.2.2.4 Integration Testing Perform integration testing on the SAtoolII program after all the component parts have been brought together into one program. In this testing use the graphical user interface to select each menu, object, and tool item at least once in a single session and check for proper operation and error handling.

5.2.2.5 Regression Testing Perform regression testing on the SAtoolII program after all the component parts have been brought together into one program. In this testing, check that each function offered in a pull-down menu in SAtool in C is offered as a method of one of the SAtoolII graphical user interface objects. Execute the method from the graphical user interface to see if the same desired result occurs as when the corresponding SAtool function is executed. Do not test the ability of SAtoolII to load or save SAtool files because the two formats are incompatible.

5.2.2.6 Stress Testing Perform stress testing at both the sub[illegible]m and the SAtoolII level. First, push the data structure limitations past the maximum values and check if the subsystems handle the failures elegantly and with adequate messages. Next, test if the SAtoolII graphical user interface adequately responds to a series of consecutive sudden, unexpected inputs from the mouse, keyboard, and files.

5.2.2.7 Acceptance Testing Follow these step to perform the acceptance testing of SAtoolII:

1. Start up SAtoolII.
2. Create a new valid three-level, seven-diagram IDEF$_0$ project; include on each diagram at least three boxes, four line segments, three labels, one squiggle, one boundary arrow, and one type of note.
3. Save the project as project *A* and exit SAtoolII.
4. Restart SAtoolII and load project *A*; check for any lost data.
5. In this same project *A*, modify the contents of one diagram, delete another diagram, and add a third diagram to the project; follow the same minimum object rules as in step 2 above.
6. Save the project as project *B* and exit SAtoolII.
7. Restart SAtoolII and load project *B*; check for any lost data.
8. Sequentially execute all selections in the PROJECT, DIAGRAM, DICTIONARY, TAXI, VIEWS, OPTIONS, and OTHER menus to check for proper operation as described in the SAtoolII User's Manual.
9. Apply each tool in the Tools window at least once to each object in the Objects window to check for proper operation as described in the SAtoolII User's Manual.
10. Exit SAtoolII.

5.2.3 Test Results The test suite was used to perform functional, module, subsystem, and stress testing on all the components of SAtoolII except for the fact utilities, fact files, error handling, complex drawing objects, and overall constraint management. These components were not implemented in this research effort. As for the components that were tested, no test failures occurred.

The integration, regression, and acceptance testing for SAtoolII are awaiting the final integration of all the components of SAtoolII into one program. Partial acceptance testing was performed on the drawing model driver program. A project was entered into it that contained five diagrams in a hierarchy that was three layers deep. No test failures occurred.

5.3 Evaluation

Because of time constraints, all the components of SAtoolII were not integrated in this research effort. Consequently, a user evaluation could not be performed. When the integration does occur in follow-on research SAtoolII should be subjected to the same user evaluation as SAtool in C. Johnson's evaluators for SAtool used the following areas for the evaluation (18:5-8–5-10):

1. System Feedback or Content of Information Displayed
2. Methods of Communication with the System
3. User Error Prevention by the System
4. Error Recovery from User Errors
5. Usefulness of the System Documentation
6. User Perception of the System versus Expectations
7. User Confidence in the System
8. Ease in Learning to Use the System
9. Manner in which Information is Displayed
10. User's Feeling of Control of the System
11. Perception of the Usefulness of the System
12. Overall Evaluation of the System

5.4 Summary

This chapter took the reader through the testing and evaluation phase of SAtoolII. The chapter began by explaining the suite of tests for SAtoolII and the results of the tests that could be performed based on the components implemented. The chapter ended by listing the areas that SAtool in C was evaluated in and suggesting that SAtoolII should be subjected to this same evaluation.

VI. Conclusions and Recommendations

6.1 Summary

This thesis contained six chapters. Chapter I introduced the reader to the thesis. Chapter II presented a literature review on the X Window System, object-oriented data modeling, and graphical user interfaces. Chapter III described the revision of the essential model and drawing model. It also described the design of the two revised models, the Ada graphical support environment, the SAtoolII graphical user interface, the IDEF$_0$ complex drawing objects, the project facts input and output, and the error handling. Chapter IV covered the implementation of some of the SAtoolII components, suggested implementations for the other SAtoolII components, and information on demonstration programs. Chapter V reviewed the testing performed on the implemented components of SAtoolII and suggested a set of criteria to be used in a user evaluation of SAtoolII when the program is completed in follow-on research. This chapter presents a summary of the thesis along with conclusions and recommendations.

6.2 Conclusions

6.2.1 Research Accomplishments The work performed by this research effort in combination with that of Kitchen (20) resulted in the following accomplishments:

- Development of a machine-independent Ada graphical support environment and demonstration of its use in the implementation of the SAtoolII graphical user interface
- Development and demonstration of an ERD to OOD mapping technique which transforms an entity-relationship model into actual Ada source code
- Demonstration of the feasibility of using Ada and object oriented design techniques in the implementation of a CASE tool

- Demonstration of the feasibility of representing the entire diagram ' ' earchy of an IDEF$_0$ model in a single program implementation
- Design and partial implementation of SAtoolII, a production-quality project editor for IDEF$_0$

6.2.2 Ada and Graphics Is it possible to put Ada and graphics in the same sentence or are the two incompatible? Before starting this thesis I probably would have taken the incompatible option. However, in the past few months I have come to appreciate the many features of the Ada language. Ada made it possible for me to implement a graphical user interface and graphical manipulation routines in a language with such admirable features as strong typing and limited variable visibility. Ada's packaging made the incremental development approach to SAtoolII much easier and manageable. Ada's type checking at compile time saved many inconsistencies and unknown type conversions that occur implicitly in lower order programming languages. At first I felt confined by Ada's string handling features, but as I became more familiar with slices and type attributes, I found that Ada took care of many of the drudgery jobs not implemented in other languages. I will admit that the compiling time and runtime size of the programs were often discouraging, but I believe that this is only a result of the implementation of the Ada language and not the language features itself. When the SAtoolII graphical user interface prototype was finally running, there was no indication that it was coded in Ada rather than C.

If I were given the task of coding another large X Window project, and had the choice of using C, C++, or Ada, I would probably choose C++. This is only because of the overhead, complexity, nd extra runtime code caused by the Ada bindings to the X Window System. As this thesis was being finished, Rational had just announced a release of the X library code of the X Window System in Ada. If this code functions as reliably and quickly as the X library in C, I would choose Ada as the development language every time.

6.2.3 SAtoolII SAtoolII probably looked like another Ada software project to Smith when he began his thesis research work in 1989 (31), but the immensity of it soon became a reality as the design of the essential and drawing models began. When he began to implement the essential model and incorporate the X Window System, the time and manpower were just not available. However, his initial work with the IDEF$_0$ models, along with Johnson's SAtool in C, and Kitchen's revision and implementation of the essential model, made it possible for me to make portions of SAtoolII a reality. If it wouldn't have been for the time needed in revising the essential model and drawing model, I probably could have brought SAtoolII up to the level of production-quality. Nevertheless, those revisions eliminated design flaws in the IDEF$_0$ model that would have later been a serious detriment to the project.

The development of SAtoolII is now at a baseline point. The components implemented so far function as planned. Nevertheless, all the components need to be implemented and the whole architecture integrated to see the full worth of the program.

6.3 Recommendations

6.3.1 SAtoolII Main Menu Selections SAtoolII has passed through the design and partial implementation phases. It still has components that need to be implemented and then integrated into a complete SAtoolII program. After this integration occurs, SAtoolII will still be wide open to enhancements. To encourage these enhancements, I have provided the internal routines, and the "hooks" in the graphical user interface for future programmers to use in improving SAtoolII. Here are my recommendations for the menu selections and program options that I did not implement:

- Print Project and Print Diagram. This menu selection should allow the user to bring up a window, choose a printer, and send the output just like in any PC-based program.
- Lay Out Project. This menu selection will take much more research, but it should enable SAtoolII to completely lay out (draw) all the diagrams and graphical objects for a project solely from the information in the essential model. In other words, an IDEF$_0$ language description of a project could be created from just a data dictionary.

- Derive Project. This menu selection will take much more research, also. It should enable SAtoolII to derive (figure out) the complete contents of the essential model (data dictionary) solely from what is in the drawing model. The user would then be free to graphically describe a system using only the $IDEF_0$ language.

- Line Rerouting. This menu selection should enable SAtoolII to reroute data lines on a diagram whose current routes are either covered or made more direct by the addition or deletion of a box.

- Dimensions. This menu selection should enable SAtoolII to improve the user's conception of a project by displaying the $IDEF_0$ diagrams in a three-dimensional view. The underlying tools are already in $IDEF_0$, the higher level logic just needs to be designed and implemented.

- Color. This feature will greatly increase the identifiability and purpose of the windows in the graphical user interface. It could also be used to enhance text and lines in a diagram, error conditions. or actions suggested by SAtoolII.

- Syntax Observance. This menu selection should enable SAtoolII to immediately inform the user if the most current addition or modification to a project violates any of the syntax rules of $IDEF_0$.

- Help Level. This menu selection should allow the help feature to cover the whole spectrum from experienced users to those needing a step-by-step tutorial.

6.3.2 SAtoolII Command Line Options In addition to this list, the inputs from the command line options are already readable by SAtoolII and need only be applied within the program. These options are the display indicator, the input file name, the background and foreground colors, the drawing font name, the help level, and the drawing line thickness.

Appendix A. *Ada Bindings to the X Window System*

A.1 Introduction

This appendix performs four tasks. First, it describes the directory location at AFIT of the Ada bindings to the X Window xlib library. It also gives some specific information on the bindings themselves. Second, it explains step-by-step how to set up an Ada X Window development environment. Third, it takes the reader through the compiling, linking, and running of a sample X Window client written in Ada. Fourth, it supplies a sample program that uses the Ada pragma interface compiler directive to call C functions.

A.2 Ada Bindings Overview

In 1987, Science Applications International Corporation (SAIC) put together an Ada source code interface (Ada bindings) to the X Window System xlib library using the Ada *pragma interface* compiler directive (15). More information on using the *pragma interface* appears at the end of this appendix.

These Ada specifications and bodies A copy of the Ada bindings source code to the xlib library was obtained by AFIT and is accessible through the sol Sun workstation or the olympus microcomputer under the /usr/X/ada/saicx2/xlib directory as shown in Figure A.1.

These Ada specifications and bodies provide the code that creates an interface between the X Window System xlib library written in C and an application program written in Ada. See below for more specific location information.

Note that under the saicx2 directory is also a xrlib directory. This directory contains Ada bindings to an X Window System resources library. This library contains functions built on top of the xlib library. The xrlib library is not available at AFIT and is different than the the X Window System Toolkit library, xtlib, that is available and is also built on top of the xlib library. SAIC is currently working on standardized Ada bindings to the xtlib.

A.3 Configuring an Ada X Window System Development Environment

Follow these steps to build your own Ada X Window System development environment on olympus or on sol:

1. Create a directory in your account and change to that directory
2. Run the Verdix Ada command 'a.mklib -i' and choose '1' at the prompt (The '-i' refers only to sol)
3. Copy the .ada files from the saicx2 directories xlib/verdix and xlib/base into your compiling directory
4. Change all the .ada files in your compiling directory to .a files
5. In the x_lib_.a file, the *pragma interface* call syntax is incorrect for the Verdix Ada compiler on sol. Change each *pragma interface* from the original syntax to the revised syntax as indicated below (Note: This job will take about 30 minutes so you may want to find someone else who has already done it):

   ORIGINAL SYNTAX:

   pragma interface (C, Ada_Function_Name, C_Function_Name);

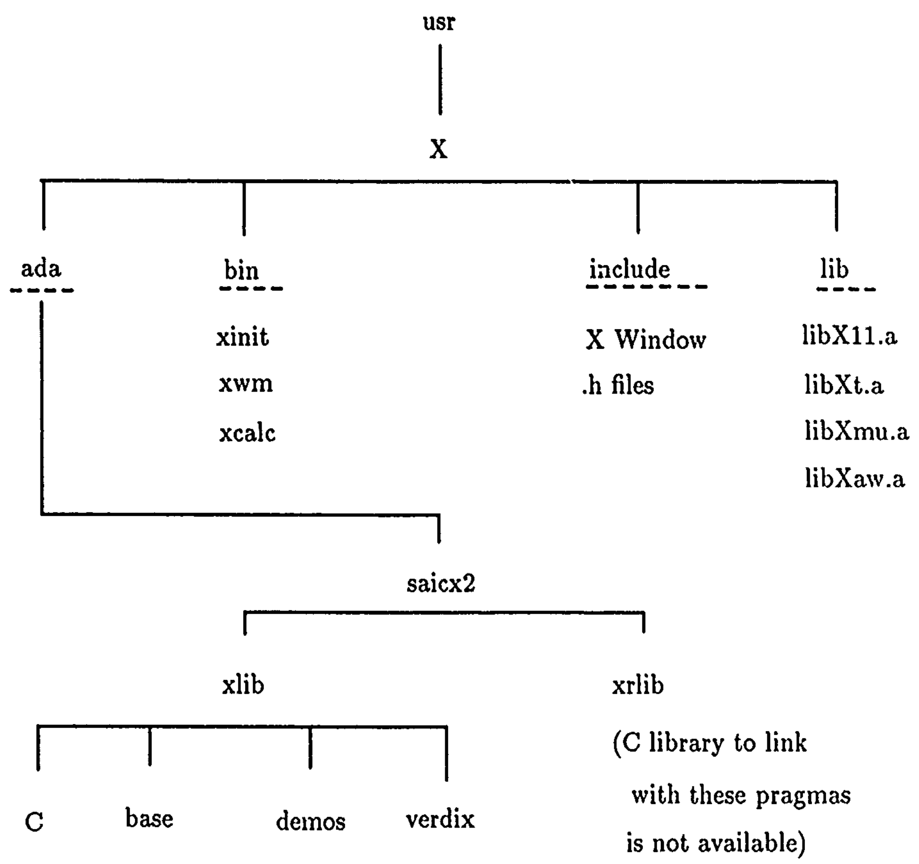


Figure A.1. X Window System and Ada Bindings Directory Tree

REVISED SYNTAX:

pragma interface (C, Ada_Function_Name);

pragma interface_name (Ada_Function_Name, C_Function_Name);

See the the pragma interface section at the end of this appendix for more information.

6. In the xlib/verdix directory is a make.inv file which lists the compilation order of the SAIC Ada files. An abridged version of that file appears below:

```
--------------------------------------------------------------
--
--  compilation order of files for Verdix version
--------------------------------------------------------------
-- The following units are assumed to be in the Verdix library:
-- UNCHECKED_CONVERSION
-- SYSTEM_ENVIRONMENT
-- SYSTEM
-- UNCHECKED_DEALLOCATION
------------------------------
-- Units at dependency level 1
------------------------------
ada x_lib_.a  -- package X_WINDOWS
------------------------------
-- Units at dependency level 2
------------------------------
ada command_line_arguments_.a
ada x_int_.a      -- package X_WINDOWS_INTERFACE
ada x_keysyms_.a  -- package KEY_SYMS
------------------------------
-- Units at dependency level 3
------------------------------
ada x_lib.a  -- package body X_WINDOWS
------------------------------
-- Units at dependency level 4
------------------------------
ada command_line_arguments.a
ada x_atoms.a    -- package body X_WINDOWS.ATOMS
ada x_colors.a   -- package body X_WINDOWS.COLORS
ada x_cursors.a  -- package body X_WINDOWS.CURSORS
ada x_cutpaste.a -- package body X_WINDOWS.CUT_AND_PASTE
ada x_events.a   -- package body X_WINDOWS.EVENTS
ada x_fonts.a    -- package body X_WINDOWS.FONTS
ada x_graphic.a  -- package body X_WINDOWS.GRAPHIC_OUTPUT
ada x_keyboard.a -- package body X_WINDOWS.KEYBOARD
ada x_regions.a  -- package body X_WINDOWS.REGIONS
ada x_win_mgr.a  -- package body X_WINDOWS.WINDOW_MANAGER
------------------------------
-- Units from VADS library
------------------------------
ada math_spec.a
ada math_body.a
```

Starting with the first file name listed above and then following down the list, compile each file using 'ada file_name' and ignore the *inline* and *pragma* warning messages. (However, do not ignore the "pragma undefined" warning. If you get this message, the pragma interface syntax is wrong for the compiler version the computer is using. A "function undefined" error will occur at load time if you do not use the proper pragma interface syntax.) The *x_events.a* file takes the longest to compile (about 10 minutes) and needs a good deal of memory. If you use the Verdix Ada compiler on the sol Sun workstation to compile x_events.a, you may not be able to have the Sun View screen interface running at the same time even if you are not at the console. If you use the Verdix Ada compiler on the olympus microcomputer to compile x_events.a, you will not have enough memory space. Instead of trying to compile the whole file at one time, break it up into these three parts and compile them as described below:

(a) event_types.a - Create a package specification called Event_Types and move all the types at the beginning of x_events.a into it. Compile the new file *event_types.a* first.

(b) x_events.a - After removing the type declarations in the beginning of this file, move the entire function called *Set_Event* out into a file of its own. Next, *with* in the Event_Types package and declare the the Set_Event procedure as *separate*. Compile the altered file *x_events.a* second.

(c) set_event.a - Move the entire *Set_Event* from x_events.a into this separate file. Compile the new file *set_event.a* third.

7. There are certain bit-level functions that are performed by C functions called by the Ada bindings. These functions are in five .c files in the saicx2/xlib/c directory. Copy these files into your compiling directory. The file names are *and.c, or.c, xbcopy.c, xbittest.c,* and *xstrlen.c.*
8. Append the five .c files listed above to a new common file called utils.c and use an editor to remove any excess comments in the file.
9. Compile utils.c to create utils.o using 'cc -c utils.c'.
10. In your UNIX path statement in your .login file, insert the /usr/X/lib, /usr/X/include, and /usr/X/bin directory paths. Ensure that these paths actually exist on the machine you are using before putting them in your .login file. You can do this by just running the UNIX change directory (cd /usr/X/lib) command and watching for a successful execution.

After about one hour total time of following the steps above to compile the Ada and C code, you will have created your own Ada development environment for the X Window System.

A.4 Compiling, Linking, and Running an X Window Client in Ada

To compile, link, and run a sample X Window client program written in Ada, do the following:

1. Copy the Hello_World.ada source code demo file from the saicx2/xlib/demos directory into your compiling directory
2. Change the extension on the file from .ada to .a
3. Compile and link the program using these commands:

```
ada Hello_World.a
a.ld Hello_World /usr/X/lib/libX11.a utils.o
cp a.out Hello_World.exe
```

4. If you are not executing an X Window Manager already, enter the 'xinit' command from the console of a Sun workstation. You should not be within the Sun View environment when you do this. (If you inadvertently execute the xinit command remotely, the X Window Server will attempt to come up on the display of the remote machine.) The xinit command looks for a .xinitrc file in your directory path. When it doesn't find it, only an empty X window screen appears. Here is a sample .xinitrc file that you can put in your directory that will enable xinit to provide a little better X Window working environment for you.

   ```
   xclock -g 50x50-0+0  -chime &
   xterm -g 80x24+0+0 -fn 9x15 &
   xterm -g 80x24+0-0 &
   twm
   ```

   This sample .xinitrc file contains actual X Window client programs that all become active UNIX processes, so be sure to end the xclock and xterms with ampersands, and leave the ampersand off the twm (Tom's Window Manager).

5. Using the mouse, move the cursor into one of the xterm windows and execute the file *Hello_World.exe*. Shortly afterwards, the outline of a window will appear. Move the mouse to position the window and click the mouse to mark the position and make the window appear. This Hello World application is rather simple; just click the mouse in the window and the phrase *HI!* will appear. To exit, type *Q* or *q* and the window will go away.

A.5 Using The Pragma Interface

The Ada pragma interface compiler directive enables an Ada program to call functions in the object code of other languages. The following is an example of how functions written in C can be called from a driver program written in Ada. The key to this process is the pragma interface statements in the Ada program. Note that in the pragma interface procedure in Ada, the C function name is preceded by an underscore.

A.5.1 Routines.c Source Code File in C

```
/* Begin Routines.c */

function1 ()
   {
   printf("This is function 1\n");
   return(1);
   }

function2 ()
   {
   printf("This is function 2\n");
   return(2);
   }

/* End Routines.c */
```

A.5.2 Driver.a Source Code File in Ada

```
package Function_Test_Package is

   function First_Function return integer;
   function Second_Function return integer;

   -- This is the revised syntax for use on sol --
   pragma interface (C, First_Function);
   pragma interface_name (First_Function, "_function1");

   pragma interface (C, Second_Function);
   pragma interface_name (Second_Function, "_function2");

end Function_Test_Package;

with Function_Test_Package;

procedure Driver is

   First_Result,
   Second_Result : integer;

   begin
   First_Result  := First_Function;
   Second_Result := Second_Function;
   end Driver;
```

A.5.3 Compiling, Linking, and Running The Program

```
cc -c Routines.c

ada Driver.a

a.ld Driver Routines.o

cp a.out Driver.exe
```

For more information, refer to page 6-2 in the Verdix User's Guide (Sun-3 UNIX version 5).

A.6 Summary

In this appendix, the reader learned about the set up and use of the Ada bindings to the X Window System xlib library. This included finding the location of the Ada bindings source code, creating an Ada X Window development environment, executing an X Window client in Ada, and examining the source code of an Ada program that used the pragma interface compiler directive to call C functions.

Appendix B. *MAGSE Reference Manual*

B.1 Introduction to the MAGSE Reference Manual

The MAGSE is the Machine-Independent Ada Graphical Support Environment. It consists of these Ada packages compiled in this order:

```
ada magse_interface.a   <-- MAGSE_Interface package specification
                            and body along with other package
                            specifications except MAGSE_Globals
ada magse_globals.a     <-- MAGSE_Globals package specification
ada window_manager.a    <-- Window_Manager package body
ada drawing_primitive.a <-- Drawing_Primitive package body
ada input_device.a      <-- Input_Device package body
ada matrix2d_stack.a    <-- Matrix2D_Stack package body
ada plane2d.a           <-- Plane2D package body
ada matrix3d_stack.a    <-- Matrix3D_Stack package body
ada pyramid3d.a         <-- Pyramid3D package body
```

The products and services offered by each of these packages are explained in the following sections.

B.2 The MAGSE_Interface Package

The MAGSE_Interface package specification provides a machine-independent interface between an Ada application program and a specific window system. It is the only package that an application needs to *with* in to use the MAGSE. This is because the MAGSE_Interface package specification contains the specifications for all the MAGSE package bodies. The MAGSE_Interface package specification contents remain unchanged for each different window system the MAGSE may be used with, but the package bodies that make up the MAGSE adapt the MAGSE subprograms into calls that the specific window system understands. Currently, the MAGSE package bodies exists only for the X Window System. Refer to Appendix A of this thesis to find out how to set up and use an Ada X Window development environment.

To make the constants, variables, procedures, and functions of the MAGSE available to an application, place the following Ada statement just before the package or procedure that will use the MAGSE resources:

```
with MAGSE_Interface;
```

B.2.1 Global Constants These are the constants that the highest level of the MAGSE_Interface makes available to an Ada application.

Max_Window_ID. This constant represents the maximum number of windows that an application can have active at one time. Reducing this number decreases the amount of memory an application requires to run in. Currently this value is 25.

Maximum_Window_String_Length. This constant represents the maximum number of characters stored for the all the non-menu character strings used in the MAGSE. Currently this value is 20.

Maximum_Menu_String_Length. This constant represents the maximum number of characters stored for menu entry character strings used in the MAGSE. Currently this value is 25.

Null_Window_String. This constant represents a non-menu Window_String containing no information.

Null_Menu_String. This constant represents a menu Window_String containing no information.

Maximum_Menu_Entries. This constant represents the maximum number of selections in a column menu. Currently this value is 10.

Zero_Menu_Entry. This constant is returned by a column menu if the user clicks outside the menu to cancel.

Main_Window_ID. This constant represents the window in an application of which all other windows are a parent of. The main window is automatically created when an application executes the Initialize_Window_Environment procedure.

Parent_Of_Main_Window_ID. This constant is used as the parameter for procedure or function calls that require the parent of the Main_Window_ID. The parent of the main window is logically the entire screen.

B.2.2 Global Types These are the types that the highest package level of the MAGSE_Interface makes available to an Ada application.

Window_ID_Type. The Window_Manager returns a variable of this type when it creates a window. Each window ID is unique. The MAGSE later requests this window ID in all window-related calls.

Window_Purpose_Type. A constant of this type names one of seven mutually-exclusive purposes that an application requests for a specific window. These seven purposes are:

- DRAWING - used for displaying on the screen general images involving text and lines
- ACKNOWLEDGE - used to display on the screen a message which a user acknowledges with a button or key press. The cursor location does not matter.
- CONFIRM - used to display on the screen a question which a user answers by clicking on a yes, no, or cancel button
- DIALOG - used to display on the screen a message which a user then responds to by entering information from the keyboard and pressing ESC to cancel or RETURN to accept
- COLUMN_MENU - used to display a vertical menu of one to ten selections which a user then clicks on to choose a selection or clicks outside of to cancel
- SIGN - used to display rectangular signs filled with text which an application can position and have the user click on
- TEXT - used to display on the screen symbols or text which serves to identify an application or region of the screen

Window_String_Type. This type maintains uniformity in the non-menu character string parameter types of the MAGSE routines. Its length is governed by the Maximum_Window_String_Length. To allow for changes in this length, the MAGSE provides a function to convert any character string type to a Window_String_Type.

Menu_String_Type. This type maintains uniformity in the menu character string parameter types of the MAGSE routines. Its length is governed by the Maximum_Menu_String_Length. To allow for changes in this length, the MAGSE provides a function to convert any character string type to a Menu_String_Type.

Menu_Entries_Range. The type represents the column menu return values ranging from the Zero_Menu_Entry through the Maximum_Menu_Entries.

Menu_Entries_List_Type. Variables of this type are an array with Maximum_Menu_Entries of character strings of Menu_String_Type. The index of this array corresponds to the selection value returned by a column menu.

Color_Table_Range. This type represents the number of indices in a color table. Currently this range is 0 to 1.

Color_Table_Type. Variables of this type are an array of color values ranging from 0 to 255. The number of entries in a color table is dictated by the Color_Table_Range.

Confirm_Type. A constant of this type names one of three values returned by a confirm window. These values are CONFIRM_YES, CONFIRM_NO, and CONFIRM_CANCEL.

Sign_Dimensions_Type. Variables of this type are passed as parameters to the Set_Sign procedure. The type is a record with four integer fields: Upper_Left_X, Upper_Left_Y, Lower_Right_X, and Lower_Right_Y, which describe the box size and location.

Sign_Record_Type. Variables of this type are handy to use by an application when it's maintaining information about the signs contained in a sign window. This type is a record with three fields:

- Dimensions : Sign_Dimensions_Type
- Sign_Name : Window_String_Type
- Name_Length : integer;

B.2.3 Global Command Line Variables These are the command line variables that the highest package level of the MAGSE_Interface makes available to an Ada application.

Acknowledge_Beep_Desired. This variable is a boolean flag indicating if the application wants a beep to occur each time the application executes the Show_Window procedure for an acknowledge window. The default value is false.

Display_Name. This variable, a Window_String_Type, contains the name of the display that the Window_Manager should get all input from and send all input to. The default value is the display of the machine the application is running on.

Input_File_Name. This variable, a Window_String_Type, contains a file name that can be used any way an application wishes. It has no meaning for the Window_Manager. The default value is Null_Window_String.

Foreground_Color_Name. This variable is a Window_String_Type containing a color name of the color the Window_Manager should use initially for all window foregrounds. Currently the available names are black and white. The default value is black.

Background_Color_Name. This variable is a Window_String_Type containing a color name of the color the Window_Manager should use initially for all window backgrounds. Currently the available names are black and white. The default value is white.

Debug_Messages_Desired. This variable is a boolean flag indicating if the Window_Manager should display the execution debug messages built into its procedures and functions. The default value is false.

Drawing_Font_Name. This variable is a Window_String_Type containing the name of a font recognized by the specific window system called upon in the Window_Manager package body. It has no meaning for the Window_Manager itself except that it is a character string. The default value is "9x15", which is a valid X Window System font name.

Program_Help_Level. This variable is an integer containing a number that an application can use any way it wishes in establishing its help levels. It has no meaning for the Window_Manager. The default value is 0.

Drawing_Line_Thickness. This variable, a positive integer, is the thickness of the lines drawn by the Set_Sign, Set_Pixel, Set_Line, and Set_Rectangle procedures. The default value is 1.

B.2.4 Constructors The highest package level of the MAGSE_Interface contains no constructors.

B.2.5 Selectors These are the selectors that the highest package level of the MAGSE_Interface makes available to an Ada application.

```
function String_To_Window_String (In_Value : in string)
                                  return Window_String_Type;

function String_To_Menu_String (In_Value : in string)
                                return Menu_String_Type;
```

B.2.6 Exceptions This is the exception raised at the highest package level of the MAGSE_Interface

```
MAGSE_Interface_Error : exception;
```

B.3 The Window_Manager Package

B.3.1 Constructors These are the constructors that the Window_Manager package of the MAGSE_Interface makes available to an Ada application.

```
procedure Initialize_Window_Environment (Main_Upper_Left_X,
                              Main_Upper_Left_Y,
                              Main_Window_Width,
                              Main_Window_Height : in integer);

procedure Terminate_Window_Environment;

procedure Create_Window (Upper_Left_X, Upper_Left_Y,
                         Window_Width, Window_Height : in integer;
                         Is_A_Pop_Up  : in boolean;
                         Purpose      : in Window_Purpose_Type;
                         Window_Name  : in Window_String_Type;
                         Parent_ID    : in Window_ID_Type;
                         Window_ID    : out Window_ID_Type);

procedure Show_Window (Window_ID : in Window_ID_Type);

procedure Clear_Window (Window_ID : in Window_ID_Type);

procedure Hide_Window (Window_ID : in Window_ID_Type);

procedure Move_Window (Window_ID : in Window_ID_Type;
                    Upper_Left_X, Upper_Left_Y : in integer);

procedure Set_Window_Purpose (Window_ID : in Window_ID_Type;
                              Purpose   : in Window_Purpose_Type);

procedure Set_Window_Font (Window_ID : in Window_ID_Type;
                           Font_Name : in string);

procedure Set_Confirm_Question (Window_ID : in Window_ID_Type;
                     Confirm_Question : in Window_String_Type);

procedure Set_Dialog_Prompt (Window_ID : in Window_ID_Type;
                    Dialog_Prompt : in Window_String_Type);

procedure Set_Menu_Entries (Window_ID : in Window_ID_Type;
                    Number_Of_Entries : in Menu_Entries_Range;
                    Entries_List : Menu_Entries_List_Type);

procedure Destroy_Window (Window_ID : in Window_ID_Type);

procedure Set_Foreground_Color (Window_ID : in Window_ID_Type;
                             Table_Index : in Color_Table_Range);

procedure Set_Background_Color (Window_ID : in Window_ID_Type;
                                Table_Index : in Color_Table_Range);
```

```
procedure Set_Border_Color (Window_ID : in Window_ID_Type;
                            Table_Index : in Color_Table_Range);

procedure Set_Color_Table (Table_Index : in Color_Table_Range;
                           R, G, B : in integer);
```

B.3.2 Selectors These are the selectors that the Window_Manager package of the MAGSE_Interface makes available to an Ada application.

```
procedure Get_Window_Limits (Window_ID : in Window_ID_Type;
                             X_Minimum, Y_Minimum,
                             X_Maximum, Y_Maximum : out integer);

procedure Top_Text_Line_Position (Window_ID : in Window_ID_Type;
                                  X_Position, Y_Position : out integer);

procedure Next_Text_Line_Position_Down (Window_ID : in Window_ID_Type;
                        Old_X_Position, Old_Y_Position : in integer;
                        New_X_Position, New_Y_Position : out integer);

procedure Next_Text_Line_Position_Up (Window_ID : in Window_ID_Type;
                        Old_X_Position, Old_Y_Position : in integer;
                        New_X_Position, New_Y_Position : out integer);

procedure Bottom_Text_Line_Position (Window_ID : in Window_ID_Type;
                           X_Position, Y_Position : out integer);
```

B.3.3 Exceptions These are the exceptions that are raised by the Window_Manager package of the MAGSE_Interface.

```
Window_Manager_Error       : exception;
Command_Line_Error         : exception;
Display_Not_Open           : exception;
No_More_Free_Windows_Error : exception;
```

B.4 The Drawing_Primitive Package

B.4.1 Constructors These are the constructors that the Drawing_Primitive package of the MAGSE_Interface makes available to an Ada application.

```
procedure Set_Cursor_Position (Window_ID : in Window_ID_Type;
                               X_Position, Y_Position : in integer);

procedure Set_Sign (Window_ID : in Window_ID_Type;
                    One_Line_Text : in string;
                    Dimensions : in out Sign_Dimensions_Type;
                    Display_Normally : in boolean);

procedure Set_Pixel (Window_ID : in Window_ID_Type;
                         X, Y : in integer;
                         Visibility_Desired : in boolean);

procedure Set_Line (Window_ID : in Window_ID_Type;
                    X_Source, Y_Source,
                    X_Destination, Y_Destination : in integer;
                    Visibility_Desired : in boolean);

procedure Set_Text (Window_ID : in Window_ID_Type;
                    X, Y : in integer;
                    Phrase : in string;
                    Visibility_Desired : in boolean);

procedure Set_Rectangle (Window_ID : in Window_ID_Type;
                         Upper_Left_X, Upper_Left_Y,
                         Lower_Right_X, Lower_Right_Y : in integer;
                         Visibility_Desired : in boolean);

procedure Set_Circle (Window_ID : in Window_ID_Type;
                      Upper_Left_X, Upper_Left_Y,
                      Lower_Right_X, Lower_Right_Y : in integer;
                      Visibility_Desired : in boolean);
```

B.4.2 Selectors This is the selector that the Drawing_Primitive package of the MAGSE_Interface makes available to an Ada application.

```
function Point_Is_In_Sign(X_Value, Y_Value : in integer;
                          Sign : in Sign_Dimensions_Type)
                          return boolean;
```

B.4.3 Exceptions This is the exception raised by the the Drawing_Primitive package of the MAGSE_Interface.

```
Drawing_Primitive_Error : exception;
```

B.5 The Input_Device Package

B.5.1 Constructors These are the constructors that the Input_Device package of the MAGSE_Interface makes available to an Ada application.

```
procedure Check_For_Change_In_Window_Already_Showing
                         (Window_ID : in Window_ID_Type;
                          Change_Found : out boolean);

procedure Wait_For_Key_Press (Key_Value : out character);

procedure Wait_For_Mouse_Click (Window_ID : out Window_ID_Type;
                                X_Position,
                                Y_Position : out integer;
                                Left_Button_Pressed,
                                      Middle_Button_Pressed,
                                      Right_Button_Pressed : out boolean);

procedure Wait_For_Key_Press_Or_Mouse_Click;

procedure Wait_For_Acknowledgement (Window_ID : in Window_ID_Type;
                                    Phrase_1 : in string;
                                    Phrase_2 : in string;
                                    Phrase_3 : in string;
                                    Phrase_4 : in string;
                                    Phrase_5 : in string);
```

B.5.2 Selectors These are the selectors that the Input_Device package of the MAGSE_Interface makes available to an Ada application.

```
function Get_Confirm_Choice (Window_ID : in Window_ID_Type)
                            return Confirm_Type;

function Get_Dialog_Response (Window_ID : in Window_ID_Type)
                             return Window_String_Type;

function Get_Menu_Entry_Choice (Window_ID : in Window_ID_Type)
                               return Menu_Entries_Range;
```

B.5.3 Exceptions This is the exception that is raised by the Input_Device package of the MAGSE_Interface.

```
Input_Device_Error : exception;
```

B.6 The Matrix2D_Stack Package

B.6.1 Constructors These are the constructors that the Matrix2D_Stack package of the MAGSE_Interface makes available to an Ada application.

```
procedure Clear_Matrix2D_Stack;

procedure Load_Matrix2D (M : in Matrix2D_Type);

procedure Load_Identity_Matrix2D;

procedure Pop_Matrix2D;

procedure Push_Matrix2D;

procedure Multiply_Matrix2D ( M : in Matrix2D_Type);

procedure Rotate_Matrix2D ( Degrees : in Float);

procedure Scale_Matrix2D ( X_Scaling, Y_Scaling : in Float);

procedure Translate_Matrix2D ( X_Translation, Y_Translation : in Float);
```

B.6.2 Selectors These are tne selectors that the Matrix2D_Stack package of the MAGSE_Interface makes available to an Ada application.

```
function Get_Matrix2D return Matrix2D_Type;

function Matrix2D_Stack_Is_Empty return boolean;

function Matrix2D_Stack_Is_Full return boolean;
```

B.6.3 Exceptions This is the exception that is raised by the Matrix2D_Stack package of the MAGSE_Interface.

```
Matrix2D_Stack_Error : exception;
```

B.7 The Plane2D Package

B.7.1 Constructors These are the constructors that the Plane2D package of the MAGSE_Interface makes available to an Ada application.

```
procedure Set_Plane2D
      (X_Minimum, Y_Minimum, X_Maximum, Y_Maximum : float);

procedure Draw_Line2D (Window_ID : in Window_ID_Type;
                      X1, Y1, X2, Y2 : in float;
                      Visibility_Desired : in boolean);

procedure Write_Text2D (Window_ID : in Window_ID_Type;
                       X1, Y1 : in float;
                       Phrase : in string;
                       Font_Size : in integer;
                       Visibility_Desired : in boolean);
```

B.7.2 Selectors These are the selectors that the Plane2D package of the MAGSE_Interface makes available to an Ada application.

```
procedure Inquire_Plane2D (X_Minimum, Y_Minimum, X_Maximum,
                          Y_Maximum : out float);

function In_Plane2D (X_Value, Y_Value : in float) return boolean;
```

B.7.3 Exceptions This is the exception that is raised by the Plane2D package of the MAGSE_Interface.

```
Plane2D_Error : exception;
```

B.8 The Matrix3D_Stack Package

B.8.1 Constructors These are the constructors that the Matrix3D_Stack package of the MAGSE_Interface makes available to an Ada application.

```
procedure Clear_Matrix3D_Stack;

procedure Load_Matrix3D (M : in Matrix3D_Type);

procedure Load_Identity_Matrix3D;

procedure Pop_Matrix3D;

procedure Push_Matrix3D;

procedure Multiply_Matrix3D (M : in Matrix3D_Type);

procedure Rotate_Matrix3D (Axis : in character; Degrees : in Float);

procedure Scale_Matrix3D (X_Scaling, Y_Scaling, Z_Scaling : in Float);

procedure Translate_Matrix3D (X_Translation, Y_Translation,
                              Z_Translation : in Float);
```

B.8.2 Selectors These are the selectors that the Matrix3D_Stack package of the MAGSE_Interface makes available to an Ada application.

```
function Get_Matrix3D return Matrix3D_Type;

function Matrix3D_Stack_Is_Empty return boolean;

function Matrix3D_Stack_Is_Full return boolean;
```

B.8.3 Exceptions This is the exception that is raised by the Matrix3D_Stack package of the MAGSE_Interface.

```
Matrix3D_Stack_Error : exception;
```

B.9 The Pyramid3D Package

B.9.1 Constructors These are the constructors that the Pyramid3D package of the MAGSE_Interface makes available to an Ada application.

```
procedure Look_At(VX, VY, VZ, PX, PY, PZ, Twist : in float);

procedure Perspective(Field_Of_View_Y, Aspect, Near, Far : in float);

procedure Draw_Line3D (Window_ID: in Window_ID_Type;
                       X1, Y1, Z1, X2, Y2, Z2 : in float;
                       Visibility_Desired : in boolean);

procedure Write_Text3D (Window_ID : in Window_ID_Type;
                        X1, Y1, Z1, X2, Y2, Z2 : in float;
                        Phrase : in string;
                        Font_Size : in integer;
                        Visibility_Desired : in boolean);
```

B.9.2 Selectors These are the selectors that the Pyramid3D package of the MAGSE_Interface makes available to an Ada application.

```
procedure Inquire_Look_At_Values(VX,VY,VZ,PX,PY,PZ,Twist : out float);

procedure Inquire_Perspective_Values (Field_Of_View_Y, Aspect, Near,
                                       Far : out float);

function In_Pyramid3D(X_Value,Y_Value,Z_Value : in float) return boolean;
```

B.9.3 Exceptions This is the exception that is raised by the Pyramid3D package of the MAGSE_Interface.

```
Pyramid3D_Error : exception;
```

Appendix C. *MAGSE Source Code Information*

C.1 Introduction

The names and contents of the Ada source files for the Machine-independent Ada Graphical Support Environment (MAGSE) are listed below. The only version of the MAGSE that is currently available interfaces with the X Window System. The source code files for the MAGSE are maintained by the Department of Electrical and Computer Engineering at the Air Force Institute of Technology.

C.2 File Names and Contents

`magse_interface.a`	`MAGSE_Interface package spec and body along with all other MAGSE package specifications except MAGSE_Globals`
`magse_globals.a`	`MAGSE_Globals package specification`
`window_manager.a`	`Window_Manager package body`
`drawing_primitive.a`	`Drawing_Primitive package body`
`input_device.a`	`Input_Device package body`
`matrix2d_stack.a`	`Matrix2D_Stack package body`
`plane2d.a`	`Plane2D package body`
`matrix3d_stack.a`	`Matrix3D_Stack package body`
`pyramid3d.a`	`Pyramid3D package body`

Appendix D. *SAtoolII Configuration Guide*

D.1 Introduction to the SAtoolII Configuration Guide

SAtoolII consists of a number of source files. In addition, the SAtoolII program uses the Machine-Independent Ada Graphical Support Environment (MAGSE) to perform its graphical user interface and drawing chores. This configuration guide details the compiling order of the MAGSE files and SAtoolII files to create the SAtoolII prototype program, the drawing model driver program, and the essential model driver program. The next section presents all of this information using a UNIX script file format to better illustrate the configuration of SAtoolII.

D.2 SAtoolII Configuration File

```
## SAtoolII Package Compiling and Incremental Program Creation Order ##

# Instructions : Execute Parts A through D, with their respective
#                levels, to produce the following executable
#                programs:
#
#    sa_prototype  - SAtoolII interface prototype and MAGSE test and
#                    demonstration program
#    dr_driver     - Drawing Model test and demonstration program
#    es_main       - Essential Model test and demonstration program

########################################################################
# Part A : Machine-independent Ada Graphics Support Environment
#          Packages (MAGSE)

# Level A-1
ada magse_interface.a  #(600 lines)
ada magse_globals.a  #(200 lines)
ada window_manager.a  #(1800 lines)
ada drawing_primitive.a  #(500 lines)
ada input_device.a  #(1300 lines)

# Level A-2
ada matrix2d_stack.a  #(300 lines)
ada plane2d.a  #(500 lines)

# Level A-3 (optional)
# Only compile these files if 3-D rendering will be used
ada matrix3d_stack.a  #(300 lines)
ada pyramid3d.a  #(700 lines)

# Level A-4
# SAtoolII interface prototype packages
ada sa_title_window.a  #(80 lines)
ada sa_help_window.a  #(90 lines)
ada sa_diagram.a  #(200 lines)
ada sa_drawing_window.a  #(100 lines)
ada sa_main_menu_window.a  #(800 lines)
ada sa_objects_window.a  #(300 lines)
ada sa_tools_window.a  #(400 lines)

# Level A-5
# SAtoolII interface prototype test and demo program "sa_prototype"
ada sa_prototype.a  #(200 lines)
a.ld sa_prototype /usr/lib/libX11.a utils.o
cp a.out sa_prototype

#########################################################################
# Part B :  Environment Types and Generic List Manager Packages

# Level B-1
ada es_genev.a

#########################################################################
# Part C : Essential Model Packages

# Level C-1
# This group of files can be compiled in any order
ada es_proj.a
ada es_activ.a
ada es_datel.a

# Level C-2
# This group of files can be compiled in any order
ada es_hista.a
ada es_conof.a
ada es_ICOM.a
ada mnuio.a

# Level C-3
ada es_calls.a

# Level C-4
ada es_factu.a

# Level C-5
ada es_esmio.a
ada es_clpwm.a

# Level C-6
# Essential Model test and demonstration program "es_main"
ada -M es_main.a
cp a.out es_main

#########################################################################
# Part D : Drawing Model Packages

# Level D-1
```

```
ada drawable.a  #(400 lines)

# Level D-2
# This group of files can be compiled in any order
ada dr_box.a  #(900 lines)
ada dr_feo.a  #(700 lines)
ada dr_footnote a  #(800 lines)
ada dr_label.a  #(800 lines)
ada dr_line.a  #(800 lines)
ada dr_metanote.a  #(700 lines)
ada dr_note.a  #(700 lines)
ada dr_squiggle.a  #(800 lines)
ada dr_stub.a  #(800 lines)
ada dr_terminator.a  #(800 lines)

# Level D-3
ada dr_diagram.a  #(1000 lines)
ada dr_project.a  #(1200 lines)
ada dr_example.a  #(1000 lines)

# Level D-4
# SAtoolII interface (text-mode derivative) packages
ada dr_title_screen.a  #(70 lines)
ada dr_help_screen.a  #(60 lines)
ada dr_drawing_screen.a  #(600 lines)
ada dr_main_menu_screen.a  #(1100 lines)
ada dr_objects_screen.a  #(100 lines)
ada dr_tools_screen.a  #(500 lines)

# Level D-5
# Drawing Model test and demonstration program "dr_driver"
ada -M dr_driver.a  #(150 lines)
cp a.out dr_driver.exe

##########################################################################
## end of file ##
```

Appendix E. *SAtoolII User's Manual*

E.1 INTRODUCTION

E.1.1 Background and Purpose The SAtoolII software is an IDEF$_0$ project editor program. IDEF$_0$ stands for ICAM (Integrated Computer Aided Manufacturing) Definition Method Zero and is a rule-based graphical symbol notation language. IDEF$_0$ was originally designed to describe the function model of a manufacturing system or environment. This function model then acted as a structured representation of the system functions and of the information and objects relating those functions. AFIT has expanded the purpose of IDEF$_0$ by using it in the requirements analysis phase of the software life cycle. SAtoolII adds the power of the computer to this requirements analysis process.

E.1.2 Features SAtoolII lets you completely create, modify, save, load, check the syntax of, and print the diagrams of an IDEF$_0$ project faster and easier than using pencil and paper or even a general-purpose computerized paint program. It supplies you with specific tools and services to build diagrams using the IDEF$_0$ language notation; however, it is not designed to teach you how to put a project together. Nevertheless, SAtoolII's built-in help facility will guide you through its many features and work with you in producing IDEF$_0$ diagrams in less than an hour.

The user interface of SAtoolII should look very familiar to you. It is based on the design of popular case tools, word processing programs, paint programs, desktop publishing programs, and circuit design software. It also offers many specialized features such as a thumbnail view of all diagrams in a project, automatic routing of lines, and the ability to store, search for, recall, and delete any diagram by name only.

E.1.3 System Requirements SAtoolII is an X Window System client program. This means that the computer having the graphical display monitor that SAtoolII appears on must have an X server and an X window manager program running on it, a keyboard, and a 3-button mouse. However, the SAtoolII program itself can run remotely on a computer networked with the computer having the graphical display, keyboard, mouse and X programs. The SAtoolII program currently can run on a computer (host or remote) with the UNIX operating system and a 68000 processor. Refer to the Getting Started section for more information.

E.1.4 Overview This manual appeals to a variety of user personalities. The anxious user can read the Getting Started section and have SAtoolII running almost immediately. The more methodical user can refer to the Getting Started section and then the Guided Tour section for a walk through the basics of SAtoolII. The curious user can study the in-depth descriptions in the Objects and Tools section and the Main Screen Menus section. The *I need it right now* user can skim the Getting Started section and the Printing a Window section to get enough information on how to produce an IDEF$_0$ diagram on paper as quickly as possible.

E.2 GETTING STARTED

E.2.1 Quick Start If you don't have time to read the manual, the steps below can get you up and running right now. The workstation you are using should have a graphical display, keyboard, 3-button mouse, and an X server and X window manager running on it. If you are missing any of these items, you will need to refer to the other sections in this manual before starting. Here are the Quick Start steps:

1. Create an SAtoolII directory and change to that directory.
2. Type 'SAtoolII RETURN' . A window outline will soon appear on the screen.
3. Use the mouse to position the window outline to a desired screen location; press the left button to finally place the window.
4. When the SAtoolII screen appears, use the left mouse button to click on PROJECT on the left side of the main menu.
5. In the PROJECT menu, use the left mouse button to click on the New Project selection.
6. When the dialog box appears, type in a new project file name of your choice and press RETURN.

You are now ready to create and modify an IDEF$_0$ diagram or multiple IDEF$_0$ diagrams. Use the left mouse button to bring down the submenu of any main menu item or to select an IDEF$_0$ object or tool; use it also to mark the start point of a drawing operation. Use the right mouse button to complete or mark the end of a drawing operation. Use the middle mouse button to cancel the drawing operation you are currently performing.

To save your work, use the left mouse button to click on PROJECT. Then use the left mouse button to click on the Save Project selection.

To exit SAtoolII, use the left mouse button to click on PROJECT. Then use the left mouse button to click on the Quit selection.

E.2.2 Operating Environment SAtoolII uses the window and graphics features of the X Window System produced by MIT. To run SAtoolII, you must have the following:

- A computer workstation or personal computer with an X server program and an X window manager running on it, a graphical display, a keyboard, and a 3-button mouse
- A host computer which runs the UNIX operating system and has a 68000 processor (The host computer can be the same as the computer workstation or personal computer)
- Directory path access to the executable version of the SAtoolII program

You can execute SAtoolII on a remote computer by using the UNIX remote shell command (rsh). Refer to the *-display* command line option for more information.

E.2.3 Command Line Options You can start the SAtoolII program by typing 'SAtoolII RETURN'. You can also follow 'SAtoolII' by a number of command line options before pressing RETURN. If you leave out one or more of these options, SAtoolII uses the default value for that option. One of the options is *-help*. This option brings up the following usage summary on the screen and terminates SAtoolII.

```
usage:  program_name [-options ...]

where options include:

  -bp or -beep                       Turn on acknowledgement beep.
                                     Default is off.

  -d [host]:server[.screen]          Set display to use.  Default is
                                     UNIX DISPLAY variable.

  -display [host]:server[.screen]    Same as above.

  -f <file_name>                     Set desired input file.
                                     Default is none.

  -file <file_name>                  Same as above.

  -fg <color>                        Set foreground color.
                                     Default is black.

  -foreground <color>                Same as above.

  -bg <color>                        Set background color.
                                     Default is white.

  -background <color>                Same as above.

  -db or -debug                      Turn on debug messages.
                                     Default is off.

  -fn <font_name>                    Set drawing font name.
                                     Default is 9x15.

  -font <font_name>                  Same as above.

  -h or -help                        Show this usage summary
                                     and exit program.
                                     Default is no summary.

  -lv <number> or level <number>     Set program help level.
                                     Default is 0.

  -ln <number> or -line <number>     Set drawing line thickness.
                                     Default is 1.
```

E.2.3.1 Acknowledgement Beep One of many special windows used by SAtoolII is an acknowledge window. This window appears on the screen with a message in it for you to acknowledge by pressing a key or mouse button. Setting the acknowledge beep on will make a beep occur each time SAtoolII display an acknowledge window.

E.2.3.2 Display One of the outstanding aspects of the X Window System is its ability to send graphical display information over a network. The display option tells SAtoolII and the X Window System the name of a remote computer and display to exchange graphical information with.

An application program that uses the X Window System ' referred to as an X client. An X client can execute on one computer and have all of its graphical input and output handled on the display, keyboard, and mouse of a remote computer connected through a network. When an X client begins execution, the X Window System automatically chooses the most efficient communication route between the client and the display. If the client executes on the same computer where the display is located, this route is rather simple. However, the UNIX remote shell command, the X network protocol, and an X server program make it possible to remotely execute an X client. Below is an example scenario on how to execute a remote X client and establish the remote connection back to the host computer:

Two UNIX workstations, each with a graphical display, are on the same network and have the same file server. The workstations are named alpha and beta. Beta runs programs three times faster than alpha. A user who logs into alpha can remotely use beta (rlogin or rsh) without a password check. The same is true for a user on beta. The user on alpha has already entered the *xinit* command and has an X window manager up on the screen. The user on beta is running another type of window system. The user of alpha wants to run SAtoolII on beta because the computer is faster, but he wants alpha to handle all the display input and output because that is the computer he is at. To do this, he enters the following commands:

```
%alpha: xhost beta
%alpha: rsh beta "SAtoolII -display alpha:0" &
```

Here is a list of important points to remember when following this example:

- An X server with an X window manager and Xterm client must be running on the workstation of the console that you are at. In the example above, that workstation is alpha.
- You enter the commands in the example above in the xterm window on your computer.
- The *xhost* tells the X server on your computer which remote computers are allowed to make connections to the X server on alpha.
- The *rsh beta SAtoolII* tells UNIX to run SAtoolII on beta's processor. Beta must have direct access to the SAtoolII executable code (symbolic links cause problems). The X server on alpha does not need to know anything about the location or purpose of the SAtoolII program. It only needs to know that SAtoolII is an X client.
- The *-display alpha:0* tells SAtoolII to receive all of its graphical input from and send all of its graphical output to display 0 on alpha. The displays attached to a single processor in UNIX are numbered starting with 0. Because alpha has only one display, the number for that display is 0.
- The ampersand tells UNIX to begin the program on beta as a separate process and return input and output functionality to the xterm window on alpha.

E.2.3.3 File Name SAtoolII saves IDEF$_0$ project information in two project files. You can use the file name option to tell SAtoolII the name of the project you want it to immediately load. SAtoolII will then load the two data files (project_name.esm and project_name.drm) for that project. Setting this option works the same as choosing the Load Project selection on the PROJECT menu.

E.2.3.4 Foreground and Background Color Each window used by SAtoolII has a foreground and background color. SAtoolII uses the background color for the background of a window and the foreground color for any text or drawings placed in the window. Setting the foreground and background options allows you to specify colors other than the default colors for the window backgrounds and foregrounds of SAtoolII.

E.2.3.5 Drawing Font Because SAtoolII uses the X Window System for its window and graphics features, it also has available to it over 100 different X font styles. SAtoolII makes several of these font styles available to you. Setting the font option works the same as choosing a font from the Drawing Font selection in the OPTIONS menu.

E.2.3.6 Usage Summary SAtoolII displays a command line usage summary any time the *-h* or *-help* option is used or if an unknown option is used. In either case, the execution of SAtoolII is always terminated.

E.2.3.7 Help Level SAtoolII offers a number of help levels to tailor its built-in help messages to the needs of the user. The help level option works the same as choosing a help level number from the Help Level selection in the OPTIONS menu.

E.2.3.8 Drawing Line Thickness SAtoolII offers several different line thicknesses for the lines used for figures in a drawing window. The line option works the same as choosing a line thickness from the Line Thickness selection in the OPTIONS menu.

E.3 A GUIDED TOUR

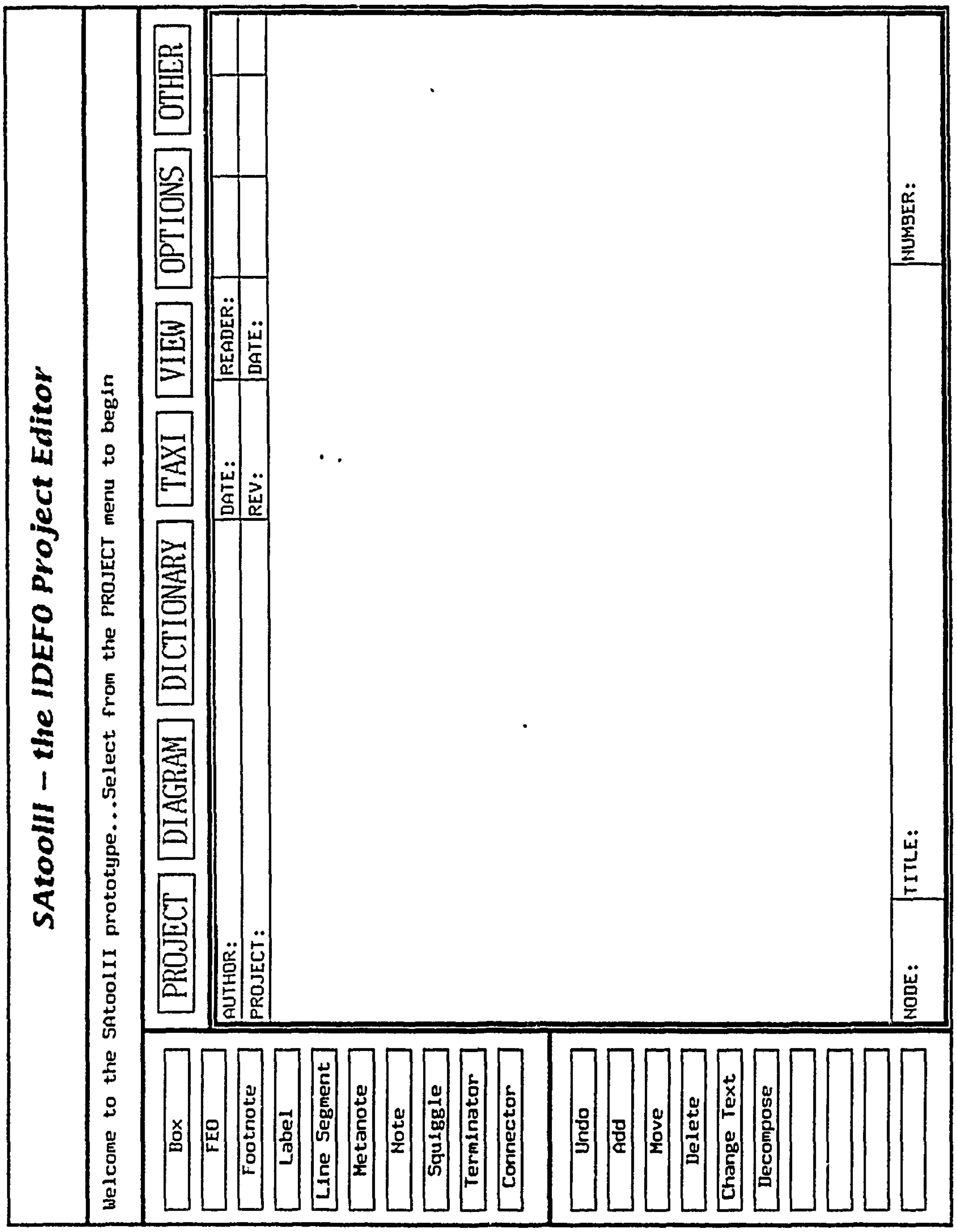
E.3.1 Introduction This section walks you through the basics of SAtoolII. The tour is designed to give you hands on experience immediately. You learn how to use some of the tools and, at the same time, how to give some simple commands and responses to the acknowledge windows, menus, confirm windows, and dialog boxes that appear on the screen.

When you finish this section you will be able to create, modify, save, and load an IDEF$_0$ project using SAtoolII. Note that the word *click* is used frequently throug! ɔut this manual. This refers to pressing and releasing a mouse button.

E.3.2 The Main Screen In the GETTING STARTED section you learned how to execute SAtoolII either on your computer or remotely. Follow the steps in that section at this time to get SAtoolII started.

Figure E.1 shows the main screen face of SAtoolII. This same screen face should be in a window on your display. The main screen has the following parts:

- Title Window. The title window is located at the top of the main screen and identifies the program as SAtoolII.
- Help Window. The help window is located just below the title window. The help window will contain a short message explaining the use or function of the menu, object, or tool you selected most recently. The help window uses a larger acknowledge window for help messages if you raise the Help Level for SAtoolII.
- Main Menu Window. The main menu window is located just below the help window. It contains the names of pull-down menus that list commands that you can give SAtoolII.
- Objects Window. The objects window is located on the left side of the main screen just below the help window. It contains a button for each of the IDEF$_0$ objects that you can place in a diagram.
- Tools Window. The tools window is located on the left side of the main screen below the objects window. The tools window contains a button for each tool that you can use to place or modify an IDEF$_0$ object in a diagram.
- Drawing Window. The drawing window is the large central work area for SAtoolII. This is where you will create and modify a diagram in an IDEF$_0$ project. In the drawing window you will see the top and bottom headers of an IDEF$_0$ diagram. The drawing window is also where SAtoolII displays pop-up windows and various views of the diagrams making up a project.
- Cursor. The cursor changes form when you move from one window to the next to remind you of the purpose of the window. In the case of the drawing window, the cursor is an arrow.



Welcome to the SAtoolII prototype...Select from the PROJECT menu to begin

Figure E.1. SAtoolII Main Screen Face

E.3.3 The Keyboard and Mouse SAtoolII accepts input from you through the keyboard and through the mouse. On the keyboard, SAtoolII recognizes only the keys that normally appear on a typewriter and the ESCAPE key. SAtoolII only uses the keyboard with a dialog box. When a dialog box is displayed you can use the keyboard to enter characters and press RETURN for SAtoolII to accept the characters or ESCAPE for SAtoolII to cancel the whole operation. SAtoolII uses the mouse for all other user input. Moving the mouse also moves the cursor on the screen. SAtoolII detects the cursor position and its movement. SAtoolII also detects when you press a button on the mouse. Click the left button to choose any button or pull-down menu or menu selection. Also use the left button to mark the start point of a drawing operation in the drawing window. Click the right button to mark the completion or endpoint of a drawing operation. Click the middle button to cancel the current drawing operation.

E.3.4 Using Objects and Tools When you begin SAtoolII, you have a project with no diagrams. The drawing window displays the IDEF$_0$ context diagram A-0. If you click on PROJECT in the main menu and then click on Load Project, SAtoolII will prompt you for a project name, load the project into memory, and display that project's context diagram in the drawing window. To add an IDEF$_0$ object to the diagram, click on an object in the objects window. Next click on the Add tool. SAtoolII now waits for you to position the cursor in the drawing window and click to mark the location of the upper left corner of the object. If you decide that you want to later change the object, click on the object button, then the Move tool button, and finally the object itself in the drawing window. SAtoolII then will wait for you once again to mark the location of the upper left corner of the object. If you make a mistake, click on the Undo tool and SAtoolII will reverse the last drawing operation you just performed. Refer to the Objects and Tools section for more information.

E.3.5 Creating and Viewing a Project SAtoolII is not a general-purpose paint program, therefore it doesn't just let you pick an object and do whatever you want with it in the drawing window. It knows enough about IDEF$_0$ to guide you in using the objects and tools properly to create IDEF$_0$ diagrams.

One example of this guidance is the Decompose tool. You can use this tool only with a box. When you select the Box button followed by the Decompose tool, SAtoolII waits for you to select a box in the current diagram. When you do, it replaces the diagram in the drawing window with the diagram describing the contents of the decomposed box. This diagram will be empty if you haven't decomposed the box yet. To return to the diagram that the box belongs to, click on the TAXI pull-down menu. Next, select Parent Diagram. SAtoolII will return the parent diagram to the drawing window.

You can be assured that anytime SAtoolII changes the diagrams in the drawing window that it has saved the contents of the current diagram in memory. To see this is true, click on VIEW in the main menu and select Project. SAtoolII will display in one view, a scaled down form of all the diagrams currently in a project. Press any key or click any button to return to the single diagram view.

E.3.6 Saving and Loading a Project To save the project you have been working on, click on PROJECT in the main menu. Select Save Project if you already gave the project a name by using New Project at the start of this session. Select Save Project As if you need to give the project a name or if you want to save the project you now have in memory under a different file name. SAtoolII will either ask you to confirm the project's file name or ask you to supply one before it continues the save operation. In either case you can cancel the save by pressing ESCAPE in the dialog window or clicking on no or cancel in the confirm window.

To load a project, click on PROJECT in the main menu and select Load Project. SAtoolII will then ask you for the name of the project. If you don't remember or if SAtoolII later says it can't find it, click on PROJECT again and use Show Directory and Change Directory to find out what your project is called and where it is stored at.

CAUTION: When SAtoolII loads a project, it destroys the project diagrams currently in memory. You can reload the project later if you have saved it, but it will be gone from memory as soon as you load another project. Don't worry too much, though. SAtoolII will ask for a confirmation from you before it destroys the project in memory.

E.3.7 Error Handling Whenever SAtoolII detects an error, it will display an acknowledge window containing an error message. If the error is not serious, SAtoolII will just wait for you to press a key or click a button to continue. If the error is serious, SatoolII will display a confirm window after you press a key or click a button. In the confirm window SAtoolII will ask if you want to continue the session or let the error go to the operating system and abort the program. When a confirm window appears for a serious error, your best move is to not abort the program. Select *NO* and then immediately try to save your project and exit SAtoolII. SAtoolII keeps track in an error file of any errors that occurred in the current session. Check this file after you exit SAtoolII if a serious error occurs. If will give you a better idea of what happened and what you can do about it.

E.3.8 Exiting SAtoolII To exit SAtoolII, first save your current project. Then click on PROJECT in the main menu and select Quit. Before exiting the program, SAtoolII will once again ask you if you want to save the project that is in memory and perform that task for before it terminates.

E.3.9 Summary This section walked you through the basics of SAtoolII. In this section you learned how to:

- Start SAtoolII
- Identify different parts of the main screen
- Use the keyboard and mouse
- Use the objects and tools
- Create and view a project
- Save and load a project
- Exit SAtoolII

To print the contents of a diagram or project refer to the section on PRINTING A WINDOW.

E.4 OBJECTS AND TOOLS

E.4.1 Introduction This section describes the objects in the objects window and the tools in the tools window that you use together to create and modify diagrams. To use a tool with a specific object you must first click on the object button and then on the tool button. After that, any subsequent clicks on a tool button will use that tool with the mcst recently designated object.

E.4.2 The Objects SAtoolII is an IDEF$_0$ project editor. It was designed to supply you with a familiar IDEF$_0$ environment to work in; hence, the purpose of the IDEF$_0$ objects in the objects window. Below is a description of each of the objects.

E.4.2.1 Box A box represents an activity on an IDEF$_0$ diagram. Along with its location in a diagram and its name, it also has many activity attributes connected with it. After clicking on DICTIONARY in the main menu, you can read these attributes by selecting Display Activity Entry or change these attributes by selecting Change Activity Field.

E.4.2.2 FEO An FEO, or for exposition only, is a special effect feature you can place in a diagram. It is not part of a diagram, but is used to illustrate the purpose of a particular action taken on the diagram. SAtoolII implements this as a character phrase that you can add to a diagram.

E.4.2.3 Footnote A footnote is the same as a note except that you can use it instead of a note if crowding in part of the diagram forces you to move the text to another location.

E.4.2.4 Label A label is how you can designate what specific data that a line segment represents. Try to position the label so the line segment it refers to is obvious. If this is not possible because of crowding in the diagram, use a squiggle to show the relationship between a label and a line segment.

E.4.2.5 Line Segment A line segment, in connection with other line segments and terminators, shows the flow of data from one box to the next. This data is identified by a label related to the line segment. SAtoolII restricts a line segment to lie either vertically or horizontally on a diagram. This means that the angle between two line segments is 90 degrees or 180 degrees. One or more line segments connected by terminators represent a data element. Along with the locations of these line segments in a diagram, the data element they represent has many attributes connected with it. After clicking on DICTIONARY in the main menu, you can read these attributes by selecting Display Data Element Entry or change these attributes by selecting Change Data Element Field.

E.4.2.6 Metanote A metanote is not part of the IDEF$_0$ description in a diagram. Instead it is observations about the diagram, such as the way it is laid out or the choice of label or box names.

E.4.2.7 Note A note lets you put nongraphical information of an analysis into a diagram. If the object that the note refers to is not obvious, use a squiggle to clarify the situation

E.4.2.8 Squiggle A squiggle is a device used when crowding on part of a diagram causes poor readability. You use it to relate a label to a line segment or a footnote marker to a line segment when you cannot place the label close enough to the object. A label has two endpoints, one near the label or footnote marker and one near the line segment.

E.4.2.9 Terminator A terminator is a symbol that you attach to the connector. The other end of the connector may be attached to a line segment, a box, or another terminator. A terminator identifies where the data is going and possibly something about where it came from. SAtoolII supplies nine kinds of terminators: arrow, boundary arrow, tunnel arrow, to-all, from-all, simple turn, junctor, dot, and null (stands for no terminator). Each time you ask SAtoolII to add a terminator to a diagram it will display the list terminators and ask you to choose from it.

E.4.2.10 Connector A connector is not an IDEF$_0$ object, at least not a visible one. It allows SAtoolII to take the tinker toy approach to building a diagram. It also allows SAtoolII to easily move multiple objects on a diagram by moving only the connectors. All connections between line segments, boxes, and terminators require a connector in between them. SAtoolII automatically puts in connectors for you. However, you can ask SAtoolII to move connectors to change the location of line segment connections, especially on the edges of boxes.

E.4.3 The Tools You can use a tool from the tools window to perform a drawing operation with an object from the objects window. Remember that SAtoolII thinks you want to use a tool with the object of the object button you most recently clicked on unless you click on another object button before clicking on the tool. Below is a description of what you can do with the tools.

E.4.3.1 Undo The undo tool reverses the most recent drawing operation that you just had SAtoolII perform with an object. If you make a drawing mistake and you want SAtoolII to undo it, this is the tool to use. As soon as you click on undo, the previous operation is reversed.

E.4.3.2 Add This tool adds an object to a diagram. After selecting the add tool, SAtoolII will wait for you to mark the position of the upper left corner of the object. If you decide later that you want to place the object in a different position, use the move tool.

E.4.3.3 Move This tool moves an object from one location in a diagram to another. SAtoolII thinks you want to move the type of object you most recently highlighted in the objects window unless you first click on another object button. After clicking on the move button, SAtoolII will wait for you to indicate which object in the diagram that you want to move. It then will wait for you to mark the new position of the upper left corner of the object.

E.4.3.4 Delete This tool removes an object from a diagram. After selecting the delete tool, SAtoolII will wait for you to indicate which object in the diagram you want to delete. This is a very powerful tool, but it is also very particular. SAtoolII will not let you use this tool to leave behind an IDEF$_0$ disaster. Consequently, deleting one object may mean deleting several objects with it to keep the diagram in order. Before SAtoolII takes such action if will ask for a confirmation from you on what it plans to do. If you use this tool to erase text, the font name setting should be the same as when the text was drawn. If the drawing window has parts of objects left in it after a delete operation, click on the Redisplay selection in the DIAGRAM menu to clean up the drawing window.

E.4.3.5 Change Text This tool lets you change the text in a note, footnote, metanote, or an FEO. After selecting the tool, SAtoolII will wait for you to indicate which object in the diagram you want to change the text of. For SAtoolII to properly erase the old text, the font name setting should be the same as when the text was drawn. If not, click on the Redisplay selection in the DIAGRAM menu to clean up the drawing window.

E.4.3.6 Decompose You can use this tool on a box only. After selecting the decompose tool, SAtoolII will wait for you to indicate which box in the diagram you want to decompose. This tool causes SAtoolII to put the current diagram aside and bring up the diagram showing the decomposition of the selected box. This diagram will be empty if you haven't decomposed the box yet. To return to the diagram where the box for the decomposition was located, click on the Parent Diagram selection in the TAXI menu.

E.4.4 Summary This section described the objects in the objects window and the tools in the tools window. In this section you learned how to use an object and tool from these windows to create and modify diagrams.

E.5 MAIN SCREEN MENUS

E.5.1 Introduction SAtoolII has seven pull-down menus on the main screen: PROJECT, DIAGRAM, DICTIONARY, TAXI, VIEW, OPTIONS, and OTHER. To pull down a menu and display its contents, click on the menu button. As you move the cursor within the menu, SAtoolII highlights the command under the cursor. To select the command, position the cursor over it until it is highlighted, then click on it. If you click outside the menu, the menu disappears and no command is selected.

Many of the commands you select in the pull-down menus will bring up dialog boxes, acknowledge windows, confirm windows, or other menus. To cancel any of these commands, do the following based on the type of window in use:

- Menu - click outside the menu
- Confirm window - click on the cancel button
- Dialog Box - press the ESCAPE key
- Acknowledge window - press any button or key because no action is ever performed

Anytime an error occurs, SAtoolII will display the error information in an acknowledge window. This type of window has no effect on the state of the project, but it will stay on the screen until you either press a key or click a button.

The rest of this section provides a detailed description of the commands listed in each pull-down menu.

E.5.2 PROJECT Menu The project menu supplies you with a list of commands you can use on an IDEF$_0$ project.

E.5.2.1 New Project The new project command erases any project information in memory and uses a dialog box to prompt you for a new project name. If a project is already in memory, it will first use a confirm window to ask if you want to destroy the project in memory only.

E.5.2.2 Load Project The load project command uses a dialog box to prompt you for the name of the project to load. If a project is already in memory, it will also use a confirm window to ask if you want to destroy that project in memory only. SAtoolII loads project information from two separate files, a file with essential model project information (*project_name.esm*) and a file with drawing model project information (*project_name.drm*).

E.5.2.3 Save Project As The save project as command uses a dialog box to prompt you for the name that you want to save a project under. If the project already exists it will use a confirm window to ask you if you want to replace the project. SAtoolII saves a project in two separate files, a file with essential model project information (*project_name.esm*) and a file with drawing model project information (*project_name.drm*).

E.5.2.4 Save Project The save project command saves the information in memory into the project files with the current project name. If you haven't designated a project name, it will use a dialog box to prompt you for a name. SAtoolII saves a project in two separate files, a file with essential model project information (*project_name.esm*) and a file with drawing model project information (*project_name.drm*).

E.5.2.5 Print Project The print project command uses an acknowledge window to tell you how to print the diagrams of a project.

E.5.2.6 Lay Out Project The lay out project command creates all the diagrams of a project completely from the essential model information contained in the *project_name.esm* file.

E.5.2.7 Derive Project The derive project command derives the contents of the essential model and data dictionary completely from the drawing model (diagram) information in the *project_name.drm* file.

E.5.2.8 Show Directory The show directory command uses an acknowledge window to display the names of the projects in the current default directory.

E.5.2.9 Change Directory The change directory command uses a dialog window to prompt you for the name of a new default directory.

E.5.2.10 Quit The quit command ends the SAtoolII program. Before ending, SAtoolII uses a confirm window to ask if you want to save the project in memory and uses a dialog box to prompt for the project name.

E.5.3 DIAGRAM Menu The diagram menu supplies you with a list of commands you can use on an IDEF$_0$ diagram.

E.5.3.1 Redisplay The redisplay command clears the drawing window and redraws all the diagram contents.

E.5.3.2 Print Diagram The print diagram command uses an acknowledge window to tell you how to print a diagram.

E.5.3.3 Clear Diagram The clear diagram command is a convenient way to remove all the objects from a diagram. SAtoolII will use a confirm window before its takes any action. It will clear a diagram only if you agree and the diagram has no boxes with decomposed diagrams associated with them.

E.5.4 DICTIONARY Menu The dictionary menu supplies you with a list of commands you can use on activity and data element entries in a data dictionary.

E.5.4.1 Show Activity Entry The show activity entry command waits for you to click on a specific box in the current diagram and then displays all the data dictionary fields of the activity entry corresponding to the box. Just press any key or click any button to return to the current diagram.

E.5.4.2 Edit Activity Field The edit activity field command waits for you to click on a specific box in the current diagram. It then uses a submenu to list all the field names of the activity entry corresponding to the box. If you click on one of the submenu entries, SAtoolII will use a dialog box to show you the current field contents and allow you to change the contents. The submenu contains the following selections: Name, Activity Number, Description, Version, Changes, Date, Author, Reference, Calls, C Number, and DRE.

E.5.4.3 Show Data Entry The show data entry command waits for you to click on a specific line segment in the current diagram. It then displays all the data dictionary fields of the data element entry corresponding to the line segment. Just press any key or click any button to return to the current diagram.

E.5.4.4 Edit Data Field The edit data field command waits for you to click on a specific line segment in the current diagram. It then uses a submenu to list all the field names of the data element entry corresponding to the line segment. If you click on one of the submenu entries, SAtoolII will use a dialog box to show you the current field contents and allow you to change the contents. The submenu contains the following selections: Name, Description, Version, Changes, Date, Author, Reference, Data Type, Minimum, Maximum, Range, and Values.

E.5.5 TAXI Menu The taxi menu supplies you with a list of commands you can use to move to other diagrams in an IDEF$_0$ project.

E.5.5.1 Context Diagram The context diagram command brings up the context diagram (the topmost project diagram) in the drawing window.

E.5.5.2 Parent Diagram The parent diagram command brings up the diagram containing the box (activity) that the current diagram was decomposed from.

E.5.5.3 Named Diagram The named diagram command uses a dialog box to prompt for a box (activity) name and brings the decomposition diagram for that box up in the drawing window if the diagram already exists.

E.5.6 VIEW Menu The view menu supplies you with a list of commands you can use to bring up different views of the diagrams in an IDEF$_0$ project other than the usual single diagram view.

E.5.6.1 Project The Project command brings up every diagram, in a scaled down form, that is in the project. Press any key or click any button to return to the single diagram view.

E.5.6.2 Path From Context The path from context command brings up every diagram, in a scaled down form, that is in the decomposition path from the context diagram through to the current diagram. Press any key or click any button to return to the single diagram view.

E.5.6.3 Children The children command brings up all the diagrams, in a scaled down form, one decomposition level below the current diagram. Press any key or click any button to return to the single diagram view.

E.5.7 OPTIONS Menu The options menu supplies you with a list of commands you can use to change user-defineable options referenced by SAtoolII to determine the user interface appearance and degree of service.

E.5.7.1 Grid The grid command lets you bring up a X/Y grid in the drawing window to help you in positioning objects. The command uses a menu so you can select on or off. The default value is off.

E.5.7.2 Drawing Font The drawing font command uses a submenu to display the names of X Window System fonts that you can uses for the text in the diagrams in the drawing window. The font name you choose stays in effect for the current session until you change it to something else. The default value is "9x15".

E.5.7.3 Line Thickness The line thickness command uses a submenu to display a choice of line thicknesses for the objects drawn in a diagram in the drawing window. This line thickness stays in effect for the current sessions until you change it to something else. The default value is 1.

E.5.7.4 Line Rerouting The line rerouting command lets you tell SAtoolII if you want it to automatically reroute any remaining line segments in the current diagram following an add, move, or delete tool operation. The purpose of the line rerouting is to ensure lines do not pass through boxes and that they take the most direct route possible within the restriction of the 90 degree and 180 degree rule. The command uses a submenu for you to select on or off from. The default value if off.

E.5.7.5 Dimensions The dimensions command lets you tell SAtoolII how you want the diagrams in the drawing window displayed, either in two-dimensions or three-dimensions. The command uses a submenu for you to select 2-D or 3-D from. The default value is 2-D.

E.5.7.6 Syntax Observance The syntax observance command lets you tell SAtoolII if you want it to check the IDEF$_0$ syntax of the current diagram in the drawing window every time you perform a drawing operation. The command uses a submenu for you to select yes or no from. The default value is no.

E.5.7.7 Help Level The help level command lets you tell SAtoolII the level of help you want it to give you. This affects the amount of help information that appears on the screen each time you click on an object, tool, or menu button. The default value is 0, which means that only the help window is used to display help information. If you use any level greater than 0, SAtoolII uses an acknowledge window to display the help information. The command uses a submenu for you to select a level from.

E.5.7.8 Warning Beep The warning beep command lets you tell SAtoolII if you want a warning beep to sound each time SatoolII displays an acknowledge window. The command uses a submenu for you to select yes or no from. The default value is no.

E.5.8 OTHER Menu The other menu supplies you with a list of commands you can use to select other miscellaneous functions offered by SAtoolII.

E.5.8.1 Check Syntax The check syntax command tells SAtoolII to check the IDEF$_0$ syntax of the current project and report the errors on the screen. This syntax checking process involves asserting facts about the project, applying a rule base to the facts, and then listing any errors. The command uses a confirm window to ask if you are sure you want to check the syntax.

E.5.8.2 Save FPT The save FPT command saves any facing page text for t' diagrams in the current project into a file with a *project_name.fpt* name. The command uses a confirm window to ask if you are sure you want to save the facing page text.

E.5.8.3 Show Stats The show stats command displays accounting figures showing how long the current session has been running. It also shows, for the current session, what buttons and menus you have selected and how many times you have selected them.

E.5.8.4 User Assessment The user assessment command uses a dialog b~~x~~ to ask you to enter information on program problems or suggestions for program improvement. [illegible] .oolII saves your input into a file with a project_name.usr name.

E.5.8.5 About SAtoolII The about SatoolII command uses an acknowledge window to display a short history and purpose for SAtoolII.

E.6 PRINTING A WINDOW

E.6.1 Introduction SAtoolII currently does not have a built-in capability to send a diagram or project to a printer or even create a file that can later be sent to a printer. But you can still easily obtain a printed copy of the IDEF$_0$ diagrams that you created using SAtoolII. This section briefly describes the X client programs used to capture a window and print it. It also tells you how to use these programs to obtain a laser printer copy of an IDEF$_0$ diagram created using SAtoolII. All the SAtoolII screen and menu illustrations in this manual were obtained using this method.

E.6.2 X Clients for Window Capturing and Printing Below is a short description of the four X client programs that are used in capturing (dumping) and printing X windows.

E.6.2.1 xwd : X Window Dump Program This program stores a window image in a specially formatted X Window dump file.

Program Command Line Options:

```
-help          Shows 'Usage:' command syntax
-nobdrs        Pixel border is not included in window dump
-out <file>    Output file name; default is standard out
-root          Makes a dump of the entire root window
```

E.6.2.2 xpr : X Window Dump Translator Program This program translates an X Window dump file into a printable output file.

Program Command Line Options:

```
-scale  <scale>    Scales bits; 3 changes 1X1 to 3X3
-height <inches>   Maximum height of window on page
-width  <inches>   Maximum width of window on page
-left   <inches>   Left margin otherwise image is centered
-top    <inches>   Top margin otherwise image is centered
-landscape         Prints image in landscape mode; default matches
                   window longest side to paper longest side
-portrait          Prints image in portrait mode; see above
-rv                Reverses foreground and background colors
-compact           Compresses white pixels on PostScript only
-output <file>     Output file name; default is standard out
-append <file>     Appends image to previously produced xpr file
-noff              Appended window appears on same page as first
-split <n>         Splits window into several pages
-device <device>   Specifies the device format to use for output.
                   For:  LN03              -device ln03
                         LA100             -device la100
                         PostScript        -device ps
                         IBM PP3812        -device pp
                         Apple LaserWriter -device lw or ps
```

Special Notes:

- The LN03 can handle windows up to 2/3 of the screen size
- LA100 pictures are always in portrait mode with no scaling
- Postscript cannot handle -append, -noff, or -split options

E.6.2.3 xdpr : X Window Dump, Translate, and Print Program This program runs the commands xwd, xpr, and lpr(1) to dump an X Window to a file, translate the file contents to a printable form, and send the translated file to a laser printer.

Program Command Line Options:

```
-filename          Specifies existing file containing xwd dump
-P<printer>        Specifies name of printer to be used
-device <device>   Specifies type of printer; see xpr options
-help              Displays list of options for xdpr

( All other options are passed to xwd, xpr, and lpr(1) )
```

E.6.2.4 xwud : X Window Undump Program This program undumps an X Window dump file into the coordinates of the original window

Command Line Options:

```
-help         Displays list of options
-in <file>    Specifies input file; default is standard input
-inverse      Undumps file in reverse video; monochrome dumps only
```

E.6.3 Printing an IDEF$_0$ Diagram and Project Follow these steps to create a printed copy of an IDEF$_0$ diagram that was created using SAtoolII.

Note: If you already have SAtoolII running, skip to step 6.

1. Have an X Window Manager running on your workstation and have two xterm windows on the screen.
2. In the first xterm window enter

   ```
   SAtoolII
   ```

3. When the window outline appears on the screen, position the SAtoolII window so that you will have access to the second xterm window.
4. Click the left mouse button to mark the position of the SAtoolII window outline and bring up the SAtoolII program.
5. Use SAtoolII to create an IDEF$_0$ diagram or load a project into SAtoolII using the Load Project selection in the PROJECT menu.
6. Bring up in the drawing window of SAtoolII the IDEF$_0$ diagram that you want a printed copy of.
7. Move the cursor into the second xterm window and enter

   ```
   xwd > diagram_name xdmp
   ```

   where diagram_name is some meaningful name for the IDEF$_0$ diagram. The xwd program copies the IDEF$_0$ diagram into an X Window dump file. When the xwd program starts up, the cursor will turn to a cross-hair.

8. Move the cross-hair inside the SAtoolII drawing window and press the left mouse button. The xwd program will beep once to stay it started storing the window in a file and beep twice when it is done.
9. Move the cursor back into the second xterm window and enter

   ```
   xpr -device ps < diagram_name.xdmp -output diagram_name.ps
   ```

   The xpr program translates the IDEF$_0$ diagram file from the X Window format to a PostScript format and puts the translated information into a new file.
10. With the cursor still in the second xterm window, enter

    ```
    lpr -Pprinter_name  diagram_name.ps
    ```

    The lpr program sends the IDEF$_0$ diagram file to the laser printer whose name is printer_name. The printing process takes about 10 minutes.

To get printed copies of all the diagrams in a project, follow steps 6 through 10 for each diagram.

Appendix F. *SAtoolII Source Code Information*

F.1 Introduction

The names and contents of the SAtoolII Ada source code files are listed below. The source code files for SAtoolII are maintained by the Department of Electrical and Computer Engineering at the Air Force Institute of Technology. Information on the Machine-independent Ada Graphical Support Environment (MAGSE) source files used by SAtoolII is in Appendix C.

F.2 Generic Manager and Environment Types File Names and Contents

es_genev.a	Multiple_Generic_Object_Manager package spec and body plus Environment_Types package (spec only)

F.3 Drawing Model File Names and Contents

drawable.a	Drawable_Class package spec and body
dr_box.a	Box_Class package spec and body
dr_feo.a	FEO_Class package spec and body
dr_footnote.a	Footnote_Class package spec and body
dr_label.a	Label_Class package spec and body
dr_line.a	Line_Class package spec and body
dr_metanote.a	Metanot_Class package spec and body
dr_note.a	Note_Class package spec and body
dr_squiggle.a	Squiggle_Class package spec and body
dr_stub.a	Connector_Stub_Class package spec and body
dr_terminator.a	Terminator_Class package spec and body
dr_diagram.a	Diagram_Class package spec and body
dr_project.a	Project_Class package spec and body

F.4 Drawing Model Demonstration File Names and Contents

dr_example.a	Example_Project package spec and body
dr_title_screen.a	Title_Screen package spec and body
dr_help_screen.a	Help_Screen package spec and body
dr_drawing_screen.a	Drawing_Screen package spec and body
dr_main_menu_screen.a	Main_Menu_Screen package spec and body
dr_objects_screen.a	Objects_Screen package spec and body
dr_tools_screen.a	Tools_Screen package spec and body
dr_driver.a	Drawing model demonstration program

F.5 Essential Model File Names and Contents

es_proj.a	Project_Class package (spec only) plus Project_Manager package spec and body
es_activ.a	Activity_Class package (spec only) plus Activity_Manager package spec and body
es_datel.a	Data_Element_Class package (spec only) plus

	Data_Element_Manager package spec and body
es_hista.a	Historical_Activity_Class package (spec only) plus Historical_Activitiy_Manager package spec and body
es_conof.a	Consists_Of_Relation_Class package (spec only) plus Consists_Of_Relation_Managor package spec and body
es_ICOM.a	ICOM_Relation_Class package (spec only) plus ICOM_Relation_Manager package spec and body
es_calls.a	Calls_Class package (spec only) plus Calls_Manager package spec and body
es_factu.a	Essential_Fact_Utilities package spec and body
es_clpwm.a	Clips_Working_Memory_Interface package spec and body
es_esmio.a	Essential_IO package spec and body
es_mnuio.a	Menu_IO package spec and body
es_main.a	Essential Model Demonstration Program

F.6 SAtoolII Interface Prototype File Names and Contents

sa_title_window.a	Title_Window package spec and body
sa_help_window.a	Help_Window package spec and body
sa_diagram.a	Diagram package spec and body
sa_drawing_window.a	Drawing_Window package spec and body
sa_main_menu_window.a	Main_Menu_Window package spec and body
sa_objects_window.a	Objects_Window package spec and body
sa_tools_window.a	Tools_Window package spec and body
sa_prototype.a	SAtoolII interface prototype main procedure

Appendix G. *Commonly Asked Ada and X Window System Questions*

G.1 Introduction

This appendix contains answers to some common Ada and X Window System questions. These answers were compiled from personal experience in implementing SAtoolII in Ada with the X Window System, from information in X Window System brochures, and from various UNIX comp.windows.x and comp.lang.ada newsgroup postings.

G.2 Ada Questions and Answers

G.2.1 Why do I get syntax errors on the pragma interface statements in the SAIC Ada code? The syntax errors are occurring in the x_int_.a file. The syntax used by SAIC for the pragma interface is not the syntax that the Verdix Ada compiler expects. To get rid of the syntax errors, change each 3-parameter pragma interface statements to two 2-parameter statements. See the example below where Ada_Function_Name and C_Function_Name are character strings.

Wrong syntax:

```
pragma interface (C, Ada_Function_Name, C_Function_Name);
```

Correct syntax:

```
pragma interface (C, Ada_Function_Name);
pragma interface_name (Ada_Function_Name, C_Function_Name);
```

G.2.2 When using the Verdix Ada compiler on a Sun workstation why do I sometimes get a "write failed" error? The error message probably reads "/:write failed, file system is full". This means that the /tmp directory on the hard disk has run out of file space. Try deleting all the files in /tmp with your userid and compile again. Verdix Ada usually cleans up these files at the end of compilation, but files can be left stranded in the /tmp directory if the compiler aborts. If deleting files in /tmp doesn't work, talk to the system admininstrator about increasing the /tmp file space allocation.

G.2.3 When using the Verdix Ada compiler on a Sun workstation why do I sometimes get a "cannot allocate more memory" error? What has probably happened is that the operating system has run out of swap space. One possible solution is to remove (kill) some of the other processes running on the machine. This will free up some of the swap space. Another possible solution is to have the system administrator increase the swap space allocation. This, in essence, increases the virtual memory of the computer.

G.2.4 When using the Verdix Ada compiler why do I get the error message "Spec of -s not found"? The error message probably reads "a.ld error: spec of -s not found in searched libraries". This error probably came up after you entered 'ada -M main_procedure.a' to create an executable program. Verdix Ada expects the main executable procedure to have the name *main_procedure*. The *main_procedure* part of the file name can be any valid Ada variable name. It must also be the name of a procedure in the main_procedure.a file.

G.2.5 How do I link the X Window System xlib into my Ada program? When you link an X client written in Ada with xlib you must have pragmas in the Ada code relating the Ada function or procedure names for xlib functions to C functions in the xlib library. One way to do this is using the Ada bindings to the X Window System developed by SAIC. Almost all the pragmas in the SAIC bindings are for functions in the X Window System; however five of them are for bit-level C utility functions that are in files that come with the SAIC bindings. You can make these functions easier to use when compiling and linking them by combining all five function files into one utilities.c file or some similar name.

Before doing any linking, all the files you plan to link together must have already been individually compiled into object code. This includes your X client application program, the xlib library, the SAIC Ada bindings, and the utilities.c file. The xlib library has already been compiled for you and the SAIC Ada bindings and utilities.c file may have already been compiled, too. (Refer to Appendix A if this is not the case.) So, you need to compile only your X client program. Now for the answer to your question. You can link your X client program, xlib, and utilities functions together into one executable program by doing the following:

```
a.ld  your_main_procedure_name  /usr/X/libX11.a  utilities.o
```

This link (load in UNIX terms) statement says that the xlib library is in a standard UNIX directory location and that the utilities object file is in your current directory. When the linking completes, your X client executable program will be in the a.out file.

G.3 General X Window System Questions and Answers

G.3.1 Where can I obtain X Window System version 11 release 4 source code files? The MIT Software Center is shipping X11R4 on four 1600bpi half-inch tapes. Call the X Hotline at (617) 258-8330 for prerecorded ordering information and a good product description.

Integrated Computer Solutions, Inc., ships X11R4 on half-inch, quarter-inch, and TK50 formats. Call 617-547-0510 for ordering information.

The Free Software Foundation (617-876-3296) sells X11R4 on half-inch tapes and on QIC-24 cartridges.

Yaser Doleh (doleh@math-cs.kent.EDU; P.O. Box 1301, Kent, OH 44240) is making X11R4 available on HP format tapes, 16 track, and Sun cartridges.

Virtual Technologies (703-430-9247) provides the entire X11R4 compressed source release on a single QIC-24 quarter-inch cartridge and also on 1.2meg or 1.44 meg floppies upon request.

Note that some distributions are media-only and do not include documents.

G.3.2 Where and how can I obtain other X Window System client source code files? You can ftp the X Window System source and other source files from the contrib directory on expo.lcs.mit.edu at MIT. To obtain these files, do the following:

1. ftp – This starts up the ftp program from the UNIX prompt.
2. open expo.lcs.mit.edu – This opens the network connection to the remote computer.
3. Enter user name of *anonymous* – Do this at the user name prompt.
4. Enter your user id as the password – Do this at the password prompt.
5. binary – This puts ftp in binary mode versus ASCII mode.

6. get file_name.tar.Z – This starts the file transfer from the remote computer to your computer. It may take a few minutes if the file is thousands of bytes in size.
7. close – This closes the network connection to the remote computer.
8. quit – This takes you out of ftp and returns you to UNIX.
9. uncompress file_name.tar.Z – This uncompresses the tar file
10. tar -xvf file_name.tar – This extracts the tar file into directories and files.

Here are some other ftp sites and the software available:

gatekeeper.dec.com	16.1.0.2	pub/X11/R4
mordred.cs.purdue.edu	128.10.2.2	pub/X11/R4
giza.cis.ohio-state.edu	128.146.8.61	pub/X.V11R4
uunet.uu.net	192.48.96.2	X/R4
crl.dec.com	192.58.206.2	pub/X11/R4
brazos.rice.edu	128.42.42.2	pub/X11R3/core.sr
charon.mit.edu	18.80.0.13	perl+patches, xdvi
cs.purdue.edu	128.10.2.1	rcs,xspee
j.cc.purdue.edu	128.210.0.3	comp.sources
nl.cs.cmu.edu	128.2.222.56	Fuzzy Pixmap 0.84
shambhala.berkeley.edu	128.32.132.54	xrn
terminator.cc.umich.edu	35.1.33.8	xscheme, msdos, atari
cayuga.cs.rochester.edu	192.5.53.209	Xfig,LaTeX styles,Jove
cfdl.larc.nasa.gov	128.155.24.55	gnu, rfc, sun, X, ucb, odu
cheddar.cs.wisc.edu	128.105.2.113	Common Lisp stuff, X11 fonts
cs.orst.edu	128.193.32.1	Xlisp
dinorah.wustl.edu	128.252.118.101	X11R3/core.src
expo.lcs.mit.edu	18.30.0.212	a home of X, portable bitmaps
gatekeeper.dec.com	128.45.9.52	X11,etc...
giza.cis.ohio-state.edu	128.146.8.61	miscellaneous similar to expo
hotel.cis.ksu.edu	129.130.10.12	XBBS, msdos, U3G toolkit
icarus.riacs.edu	128.102.64.1	SLIP, chkpt, macdump, Xpostit
interviews.stanford.edu	36.22.0.175	InterViews X toolkit
jpl-mil.jpl.nasa.gov	128.149.1.101	Tex, Mac, Gnu, Xv11R2,3
m9-520-1.mit.edu	18.80.0.45	Xim (X image viewer)
mordred.cs.purdue.edu	128.10.2.2	X11R3
polyslo.calpoly.edu	129.65.17.1	src/spaceout.tar.Z for X11
scam.berkeley.edu	128.32.138.1	X sources, etc.
sun.soe.clarkson.edu	128.153.12.3	X11 fonts, TeX
think.com	10.4.0.6	X11.2 Interviews 3d
vaxa.isi.edu	128.9.0.33	X, db
wheaties.ai.mit.edu	128.52.32.13	tX11
xanth.cs.odu.edu	128.82.8.1	comp.srcs

G.3.3 Where can I find books and articles on X that are good for beginners? Ken Lee of the DEC Western Software Laboratory (klee@wsl.dec.com) regularly posts to comp.windows.x and ba.windows.x a list of reference books and articles on X and X programming. Here is an unordered set of useful reference books and tutorials, most of which appear on that list [comments are gathered from a variety of places and are unattributable]:

Jones, Oliver, "Introduction to the X Window System," Prentice Hall, 1989. A fine introduction to programming with Xlib; fairly good background to the X protocol; nice discussion of Xlib, the X library. ISBN 0-13-499997-5.

Young, Doug. "The X Window System: Applications and Programming with Xt (Motif Version)," Prentice Hall, 1989 (ISBN 0-13-497074-8). The excellent tutorial "X Window Systems Programming and Applications with Xt," (ISBN 0-13-972167-3) updated for Motif. [The examples from the Motif version are available on expo in ftp/contrib/young.motif.tar.Z]

Scheifler, Robert, James Gettys, and Ron Newman, "X Window System: C Library and Protocol Reference," Digital Press, 1988. The bible on X. This is the most complete published description of the X programming interface and X protocol. It should not be one's first book on X, though. ISBN 1-55558-012-2. DP order number EY-6737E-DP.

Nye, Adrian, "Xlib Programming Manual, Volume 1" and "Xlib Reference Manual, Volume 2," O'Reilly and Associates, 1988. A superset of the MIT X documentation; the first volume is a tutorial with broad coverage of Xlib, and the second contains reference pages for Xlib functions and many useful reference appendices. ISBN 0-937175-26-9 (volume 1) and ISBN 0-937175-27-7 (volume 2).

Nye, Adrian, and Tim O'Reilly, "X Toolkit Programming Manual, Volume 4," O'Reilly and Associates, 1989. The folks at O'Reilly give their comprehensive treatment to programming with the MIT X11R3 Intrinsics; some information on X11R4 is included.

O'Reilly, Tim, ed., "X Toolkit Reference Manual, Volume 5," O'Reilly and Associates, 1989. A professional reference manual for the MIT X11R3 Xt; some information on X11R4 is included.

G.3.4 What do these X Window System abbreviations mean? The following are abbreviations often seen in X Window System literature.

Xt: The X Toolkit Intrinsics is a library layered on xlib which provides the functionality from which the widget sets are built. An *Xt-based* program is an application which uses one of those widget sets and which uses Intrinsics mechanisms to manipulate the widgets.

Xmu: The Xmu library is a collection of miscellaneous utility functions useful in building various applications and widgets.

Xaw: The Athena Widget Set is the MIT-implemented sample widget set distributed with X11 source since X11R2.

Xm: The OSF/Motif widget set is from the Open Software Foundation; binary kits are available from many hardware vendors.

XUI: DEC's X-programmer's toolkit, including a widget set and a high- level widget description language, is being phased out.

Xhp (Xw): The Hewlett-Packard Widget Set was originally based on R2++, but several sets of patches exist which bring it up to R3, as it is distributed on the X11R4 tapes.

G.3.5 How can I get an X Window System server on a PC? The following are good X server sources:

Locus Computing (800-955-6287; CA: 213-670-6500; UK: +44 296 89911) has a server called PC-Xsight which also appears in Acer's X terminal.

HP (800-752-0900) has the "HP Accelerated X Window Display Server" (HP AXDS/PC; HP part D2300B) which will run on any AT-class DOS machine with 640KB, MSDOS 3.1 or higher, and the HP Intelligent Graphics Controller 10 card, to which the X11R3-based server is downloaded (avoiding performance-limitations from PC RAM-size and processor speed).

Graphic Software Systems (GSS) (503-641-2200) makes PC-Xview, an MSDOS-based X server which interfaces with PC/TCP Plus networking software from FTP Software and Excelan's LAN WorkPlace for DOS. The server works with (a) 286, 386, 486 (b) EGA, VGA, DGIS displays. (c) DOS 3.2 and above (d) Microsoft, Logitech, Mouse Systems Mice (e) 640k memory up to 16 MB memory [the PC-Xview/16 is available for PCs with extended memory].

VisionWare's XVision is a Microsoft Windows-based X server which allows an IBM-compatible PC or PS/2 to display X clients running on a networked computer at the same time as local DOS programs. VisionWare is at 612-377-3627.

Integrated Inference Machines (714-978-6201 or -6776) is shipping X11/AT, an X server that runs under MS-windows. The server converts an IBM-AT into an X terminal which can simultaneously run MS-DOS and Microsoft Windows applications.

Hummingbird Communications (Canada 416-470-1203) produces the HCL-eXceed and HCL-eXceed Plus for EGA, VGA, and VGA+ controllers.

PC DECwindows a.k.a. the PC DECwindows Display Facility is an MS-DOS application that turns your PC into an X11R3 terminal. It supports DECnet. Available from DEC.

Pericom's TeemTalk-X for IBM clones allows toggling between X and DOS.

G.3.6 Where can I obtain an X Window System paint/draw program? Here are some better-known ones:

tgif is a 2-D drawing tool using only xlib. It is available on expo.

xpic is an object-oriented drawing program. It supports multiple font styles and sizes and variable line widths; there are no rotations or zooms. xpic is quite suitable as an interactive front-end to pic, though the xpic-format produced can be converted into PostScript. (The latest version is on the R4 contrib tape in clients/xpic.)

xfig is an object-oriented drawing program supporting compound objects. The text-handling is limited. The xfig-format can be converted in PostScript or other formats. One version is on the R4 contrib tape in clients/xfig; it is one of the several 'xfig' programs which several groups independently developed parallel versions of from the R3 xfig.

idraw 2.5 supports numerous fonts and various line styles and arbitrary rotations. It supports zoom, scroll, color draws, and fills.

dxpaint is a bitmap-oriented drawing program most like MacPaint; it's good for use by artists but commonly held to be bad for drawing figures or drafting.

ArborText (313-996-3566) offers PubDraw, an X11-based drawing program, on Sun, HP and Apollo workstations.

G.3.7 Where can I get a PostScript previewer for the X Window System? Here are some suggested previewers:

xps is available from almost everywhere that the X11 contributed source can be found. The version currently on expo is based on Crispin Goswell's PostScript interpreter with fixes and speedups by John Myers and Barry Shein and an X11 driver by Terry Weissman. There are known problems with fonts. The package is good for lowering the edit-print-edit cycle in experimenting with particular PostScript effects.

Ghostscript is distributed by the Free Software Foundation (617-876-3296) and includes a PostScript interpreter and a library of graphics primitives. The README for the current version, 1.3, points out that it doesn't take advantage of many of the facilities offered by X but that this is intended to change in the future. The software can probably be found on prep.ai.mit.edu. A 1.4beta may be found on uunet. [2/90]

ScriptWorks is Harlequin's software package for previewing and printing PostScript(R) descriptions of text and graphics images; previewers for X are available. For information call +44-223-872522 or send email to scriptworks-request@uk.co.harlqn.

Digital's dxpsview runs on UWS 2.1 and 2.2.

Sun's pageview runs with the X11/NeWS server.

G.3.8 How do I convert Mac/TIFF/GIF/Sun/PICT/Face/img/FAX/etc images to the X Window System format? The likeliest program is an incarnation of Jef Poskanzer's useful++ Portable Bitmap Toolkit, which includes a number of programs for converting among various image formats. It includes support for many types of bitmaps, gray-scale images, and full-color images. The latest version, PBMPLUS, was posted to the net about 11/22/89; it is also on the R4 tape under contrib/clients/pbmplus. Useful for viewing some image-formats is Jim Frost's xloadimage, a version of which is in the R4 directory contrib/clients/xloadimage.

G.3.9 Where can I obtain other language bindings than C to the X Window System libraries? Versions of the CLX Lisp bindings are part of the X11R3 and X11R4 core source distributions. The latest version of CLX (R4.1) is available from expo for ftp as contrib/CLX.R4.1.tar.Z [Chris Lindblad (cjl@AI.MIT.EDU), 4/90]; this version fixes bugs reported against the R4 distribution.

Ada bindings were written by Mark Nelson and Stephen Hyland at SAIC for the DoD. The bindings can be found on hapo.sei.cmu.edu or on wsmr-simtel20.army.mil and are also in the Ada Software Repository (ASR). R3 bindings should be available by the end of 1/90.

Prolog bindings (called "XWIP") written by Ted Kim at UCLA while supported in part by DARPA are available by FTP on MIT's expo in the xwip.tar.Zl file. These prolog language bindings depend on having a Quintus-type foreign function interface in your prolog. The developer has gotten it to work with Quintus and SICStus prolog. Inquiries should go to xwip@cs.ucla.edu.

GHG is developing X bindings and a complete Ada re-implementation of X; check Lionel Hanley at 713-488-8806.

Rational announced in October 1990 that it had placed an Ada reimplementation of the MIT C-based Xlib on expo.lcs.mit.edu. The two files are:

```
contrib/ada.xlib.README        4kb
contrib/ada.xlib.tar.Z         2.9Mb
```

This Ada version of Xlib is compatible with X11R4. It is a beta release with the release number of 560. The sources are about 10MB when they are restored from the tar file, of which 3MB is PostScript documentation. Makefiles and Imakefiles are also included. This is a complete reimplementation of Xlib; it is not just an Ada skin over the C libraries. The Xlib compiles and runs with the Rational native R1000 Ada compiler (Delta1) and Delta2 versions and with the TeleSoft TeleGen2 68K UNIX compiler (version 1.4) under SunOS (version 4.0.3 or later).

G.4 X Window System Programming Questions and Answers

G.4.1 Why do I get an error message that my display does not open when I run an X client program? This error message may have been output by the X Window System or the X client program. What has happened is that you have attempted to open the display of an X Window client on a computer that either does not have a display monitor or the computer with the display monitor doesn't have an X server running on it. Before giving you some solution ideas, here is some background information to help you better understand what the X Window System is up to. One of the features of the X Window System is the ability to have an X client execute on one computer on a network and perform all of its display input and output on another computer with a graphical display. If you don't designate an alternate display when you first execute an X client, the X Window System automatically uses the display monitor of the computer it is running on. When such a monitor doesn't exist or if no X server is running on the computer that is supposed to receive and send display information, then the X client will not be able to open a display. To solve the problem, either run the X client program from the console keyboard of the machine that has a graphical display monitor and an X server running on it. Otherwise, check [illegible] X client program has a *-display* command line option which allows you to designate an alternate display for the X client to use. To get an X server running on a computer with a graphical display monitor, enter 'xinit' from the console keyboard of the computer.

G.4.2 What is the difference between a Screen and a screen? The *Screen* is an X Window System Xlib structure which includes the information about one of the monitors or virtual monitors which a single X display supports. An X server can support several independent screens. They are numbered unix:0.0, unix:0.1, unix:0.2, etc; the *screen* or *screen_number* is the second digit (0, 1, 2, etc.) which can be thought of as an index into the array of available Screens on the particular Display connection.

The X functions which you can use to obtain information about the particular Screen on which your application is running typically have two forms - one which takes a Screen and one with takes both the Display and the screen_number. In Xt-based programs, you typically use XtScreen(widget) to determine the Screen on which your application is running, if it uses a single screen. (Part of the confusion may come from the fact that some of the functions which return characteristics of the Screen have *Display* in the names - XDisplayWidth, XDisplayHeight, etc.)

G.4.3 Why doesn't anything appear when I run this simple X client program?

```
...
the_window = XCreateSimpleWindow(the_display,
     root_window,size_hints.x,size_hints.y,
     size_hints.width,size_hints.height,BORDER_WIDTH,
     BlackPixel(the_display,the_screen),
     WhitePixel(the_display,the_screen));
...
XSelectInput(the_display,the_window,ExposureMask|ButtonPressMask|
        ButtonReleaseMask);
XMapWindow(the_display,the_window);
...
XDrawLine(the_display,the_window,the_GC,5,5,100,100);
...
```

You are correct to map the window before drawing into it. However, the window is not ready to be drawn into until it actually appears on the screen - until your application receives an expose event. Drawing done before that will generally not appear. You'll see code like this in many programs; this code would appear after a window was created and mapped:

```
while (!done)
  {
    XNextEvent(the_display,&the_event);
    switch (the_event.type) {
      case Expose:     /* On expose events, redraw */
              XDrawLine(the_display,the_window,the_GC,5,5,100,100);
              break;
      ...
      }
  }
```

Note that there is a second problem: some X servers don't set up the default graphics context to have reasonable foreground or background colors, and your program should not assume that the server does, so this program could previously include this code to prevent the case of having the foreground and background colors the same:

```
...
the_GC_values.foreground=BlackPixel(the_display,the_screen);
the_GC_values.background=WhitePixel(the_display,the_screen);
the_GC = XCreateGC(the_display,the_window,
              GCForeground|GCBackground,&the_GC_values);
...
```

Note that the code uses BlackPixel and WhitePixel to avoid assuming that 1 is black and 0 is white or vice-versa. The relationship between pixels 0 and 1 and the colors black and white is implementation-dependent. They may be reversed, or they may not even correspond to black and white at all.

G.4.4 Why doesn't my program get the keystrokes I select for? The window manager controls how the input focus is transferred from one window to another. In order to get keystrokes, your program must ask the window manager for the input focus. To do this, you must set up what are called *hints* for the window manager. If your applications is Xlib-based, you can try something like this:

```
XWMHints wmhints;
...
wmhints.flags = InputHint;
wmhints.input = True;
XSetWMHints(dpy, window, &hints);
```

Appendix H. *IDEF$_0$ Drawing Model Project File Format*

H.1 Project File Format

Listed below is the file format for the project file saved by the drawing model demonstration program.

```
IDEFO Drawing Model Project File
Project Name     :
File Created On  : 11/11/90 15:19:34
====================================================
*DIAGRAM* (A-0)
*********
NAME: <character string>
VERSION: <character string>
LAST CHANGED: <character string>
C NUMBER: <character string>
RELATED PARENT BOX: <character string>
--DRAWABLE TABLE--
FEO ROW            : <character string 1> ... <character string 25>
NOTE ROW           : <character string 1> ... <character string 25>
FOOTNOTE ROW       : <character string 1> ... <character string 25>
METANOTE ROW       : <character string 1> ... <character string 25>
SQUIGGLE ROW       : <character string 1> ... <character string 25>
LABEL ROW          : <character string 1> ... <character string 25>
LINE SEGMENT ROW   : <character string 1> ... <character string 25>
BOX ROW            : <character string 1> ... <character string 25>
TERMINATOR ROW     : <character string 1> ... <character string 25>
CONNECTOR STUB ROW: <character string 1> ... <character string 25>
*FEO*
*****
NAME: <character string>
VERSION: <character string>
LAST CHANGED: <character string>
X POSITION: <integer>
Y POSITION: <integer>
PARENT DIAGRAM: <character string>
PICTURE: <character string>
*NOTE*
******
NAME: <character string>
VERSION: <character string>
LAST CHANGED: <character string>
X POSITION: <integer>
Y POSITION: <integer>
PARENT DIAGRAM: <character string>
CONTENTS: <character string>
*FOOTNOTE*
**********
NAME: <character string>
```

```
VERSION: <character string>
LAST CHANGED: <character string>
X POSITION: <integer>
Y POSITION: <integer>
PARENT DIAGRAM: <character string>
CONTENTS: <character string>
UPPER LEFT X MARKER: <integer>
UPPER LEFT Y MARKER: <integer>
RELATED SQUIGGLE: <character string>
*METANOTE*
**********
NAME: <character string>
VERSION: <character string>
LAST CHANGED: <character string>
X POSITION: <integer>
Y POSITION: <integer>
PARENT DIAGRAM: <character string>
CONTENTS: <character string>
*SQUIGGLE*
**********
NAME: <character string>
VERSION: <character string>
LAST CHANGED: <character string>
X POSITION: <character string>
Y POSITION: <character string>
PARENT DIAGRAM: <character string>
X OTHER POSITION: <integer>
Y OTHER POSITION: <integer>
RELATED LINE SEGMENT: <character string>
RELATED FOOTNOTE: <character string>
RELATED LABEL: <character string>
*LABEL*
*******
NAME: <character string>
VERSION: <character string>
LAST CHANGED: <character string>
X POSITION: <integer>
Y POSITION: <integer>
PARENT DIAGRAM: <character string>
TEXT: <character string>
FOR A DATA ELEMENT: <TRUE | FALSE>
RELATED SQUIGGLE: <character string>
RELATED DATA ELEMENT: <character string>
RELATED CALL: <character string>
*LINE SEGMENT*
**************
NAME: <character string>
VERSION: <character string>
LAST CHANGED: <character string>
X POSITION: <integer>
Y POSITION: <integer>
```

```
PARENT DIAGRAM: <character string>
RELATED START CONNECTOR STUB: <character string>
RELATED END CONNECTOR STUB: <character string>
RELATED SQUIGGLE: <character string>
RELATED DATA ELEMENT: <character string>
*BOX*
*****
NAME: <character string>
VERSION: <character string>
LAST CHANGED: <character string>
X POSITION: <integer>
Y POSITION: <integer>
PARENT DIAGRAM: <character string>
DRE: <character string>
RELATED DECOMPOSITION DIAGRAM: <character string>
RELATED ACTIVITY:
--ICOM CONNECTIONS TABLE--
INPUT ROW    : <character string 1>...<character string 10>
CONTROL ROW  : <character string 1>...<character string 10>
OUTPUT ROW   : <character string 1>...<character string 10>
MECHANISM ROW: <character string 1>...<character string 10>
*TERMINATOR*
************
NAME: <character string>
VERSION: <character string>
LAST CHANGED: <character string>
X POSITION: <integer>
Y POSITION: <integer>
PARENT DIAGRAM: <character string>
SYMBOL: <terminator symbol>
DIRECTION: <terminator direction>
ICOM CODE: <ICOM connection>
RELATED CONNECTOR STUB: <character string>
*CONNECTOR STUB*
****************
NAME: <character string>
VERSION: <character string>
LAST CHANGED: <character string>
X POSITION: <integer>
Y POSITION: <integer>
PARENT DIAGRAM: <character string>
RELATED LINE SEGMENT: <character string>
RELATED BOX: <character string>
RELATED TERMINATOR: <character string>
=====================================================
*DIAGRAM* (A0)
*********
. . . . . . .
=====================================================
*DIAGRAM* (An)
*********
```

```
. . . . . . .
=================================================================
*END OF FILE*
```

H.2 Explanation of the Project File and Internal Data Structure

The drawing model file is a copy of the contents of the drawing model internal memory data structure. The drawing model data structure is a variant record. The record contains fields common to all drawable objects and fields unique to each drawable object. The common fields are the first seven fields (NAME..PARENT DIAGRAM) listed in the file format above for each of the drawable objects. The records and fields in the drawing data model data structure are mapped directly from the drawing model diagram: drawable objects became records, attributes of objects became simple record fields, 1-to-1 and 1-to-M relationships became matrix record fields.

Most of the fields in the drawing model data structure are either a character string, an integer, or a boolean data type. However, certain fields contain specific enumerated types. These are described below:

```
<ICOM connection> == INPUT | CONTROL | OUTPUT | MECHANISM

<terminator symbol> == ARROW | BOUNDARY_ARROW | TUNNEL_ARROW | TO_ALL |
                       FROM_ALL | SIMPLE_TURN | JUNCTOR | DOT | A_NULL

<terminator directio.˜ ==

   NONE | LEFT | RIGHT | UP | DOWN |          -- ARROW and SIMPLE_TURN
   INPUT | CONTROL | OUTPUT | MECHANISM |     -- BOUNDARY_ARROW
   HIDDEN_SOURCE | HIDDEN_DESTINATION |       -- TUNNEL_ARROW
   TO_ALL | FROM_ALL |                        -- TO_ALL, FROM_ALL
   RIGHT_UP | LEFT_UP |
   RIGHT_DOWN | LEFT_DOWN |                   -- JUNCTOR (horizontal_curve)
   UP_RIGHT | UP_LEFT | D
   OWN_RIGHT | DOWN_LEFT                      -- JUNCTOR (vertical_curve)
```

In addition to the fields resulting from the data model mapping, the drawing model data structure also contains record fields used to implement a diagram hierarchy. Each of the diagrams (either in memory or in a file) is a node in a project tree. The root of the tree is the context diagram which always exists; all other diagrams are optional. When a box in a diagram is decomposed, this results in the creation of a leaf, that is, a decomposition diagram. Intermediate nodes in the project tree are decomposition diagrams whose one or more boxes have also been decomposed. Each box in a diagram has either none or one unique decomposition diagram. The project tree is doubly-linked. each diagram knows who its children diagrams are and each child diagram knows who its parent diagram is. This is all accomplished through the relationship set up between boxes and diagrams.

The doubly-linked project tree allows a driver program to move among the diagrams in the tree in order to display, add, delete, modify, and save diagrams. The diagrams are saved to a file in a prefix tree traversal format with the context diagram saved first and its lowest numbered box saved next.

Appendix I. *Executive Overview*

I.1 Introduction

Entity-relationship diagrams (ERDs) and object-oriented design (OOD) have recently caught the attention of graphical user interface and CASE tool programmers. They have been attracted to ERDs because of their use in modeling the objects, the object attributes, and the inter-object relationships of a system. They have been drawn to OOD because of its use in the building block approach to user interface design and in its encapsulation of data structures with methods. Ada offers constructs such as packaging, information hiding, and strong typing that provide the implementation fundamentals for ERD modeling and OOD; however Ada has not been the language of choice for graphics programming. One drawback has been the lack of an adequate technique for mapping ERD diagrams to actual Ada source code. Another drawback has been the cumbersome text input and output offered for Ada and the lack of a comprehensive Ada graphical support environment. This paper addresses solutions to these issues. First, it offers a six-step process for transforming an entity-relationship model of a system into actual Ada source code and tells the results of using this process in the transformation of the $IDEF_0$ drawing model. Second, it describes the design of the machine-independent Ada graphical environment (MAGSE) and its implementation using Ada pragma interface calls to the C-based xlib library of the X Window System. Third, it summarizes how the the drawing model and the MAGSE serve as components in the partial implementation of a CASE tool called SAtoolII.

I.2 ERD-to-OOD Transformation

Entity-relationship diagrams provide a method for identifying the entities of a system, the attributes of those entities, and the entity interrelationships. After ERDs are created to model a system, a six step process can be followed to transform the contents of the ERDs into actual Ada source code.

In step one, create an entity-relationship model of a system to identify the entities, attributes, and relationships in the system. Label all relationships as one-to-one, one-to-many, or many-to-many. In step two, look at the entities as objects and consider the implementation of the system using these objects. Determine if these objects completely identify all the objects of the system being modeled. In step three, look at the many-to-many relationships as objects and consider the implementation of them as correlation table objects. For the many-to-one relationships consider putting a referencing attribute field in the *1* object. For the one-to-one relationships, consider which object should have the referencing attribute field. In step four, modify the entity-relationship model to reflect the lessons learned from steps two and three. Continue iterating through steps two and three until the entities, attributes, and relationships completely model the system. To test the system, use *what if* situations that the system should be able to handle, and walk through the model to check if the entities, attributes, and relationships *model* the particular situation. In step five, code each entity and many-to-many relationship of the model as an object. Code the descriptive attributes of relationships or entities as record fields in each object. Include in the many-to-many relationship data structures a relationship tuple containing referencing attributes. Assign each object a type class package, an object manager package, and an input/output package. In the manager and input/output packages, *with* in the types class package.

In step six, create constructor procedures in the object manager package to set descriptive attributes and referencing attributes for entities on the *one* side of a one-to-many or one-to-one relationship. Create selector functions to retrieve attributes and all relationships. The constructors and selectors should completely represent all the visible routines from an object manager package.

The types utilized in the parameters for these routines should completely represent all visible types from an object manager package. This completeness and visibility will be correct if the entity-relationship model completely contains all entity attributes and entity relationships. If any additional *visible* constructors or selectors need to be added to an object manager package, examine the entity-relationship model to see if a corresponding attribute or relationship exists for the new routine. If one does not exist, one of the following is probably true:

- The routine is a generalization of another currently visible routine
- The routine corresponds to an attribute or an entity relationship that was previously overlooked
- The routine has been incorrectly made visible and should only appear in the body of the object manager package

This process of tranforming an entity-relationship model of a system into Ada source code was applied to the drawing model of IDEF$_0$. IDEF$_0$ is the ICAM Definition Method Zero graphical notation language adopted by the U.S. Air Force to produce a function model of a manufacturing system or environment (23:1-1). The Air Force Institute of Technology (AFIT) is conducting research in the use of IDEF$_0$ in the requirements analysis phase of the software lifecycle and its use in a CASE tool. Researchers at AFIT have divided the modeling of IDEF$_0$ into two models, an essential model and a drawing model. The essential model involves those parts of IDEF$_0$ that represent the semantics of the language and include such things as activities and data elements. The drawing model comprises the graphical constructs used to represent the particular IDEF$_0$ analysis such as boxes and line segments (31:2-7).

When the six-step transformation process was applied to the IDEF$_0$ drawing model, it resulted in an autonomous packaging of the eleven drawing objects (entities) in the model: diagram, box, FEO (for exposition only), footnote, label, metanote, note, squiggle, connector, terminator, and line segment. The autonomy between the packages came about partly because of the transformation process and partly because of the desire to discourage any coupling. Each of the packages has no procedure or function coupling with the other packages. All ties among the various drawing objects are maintained through the use of drawing object name strings. Because the diagram object maintains a list of the drawing objects that are on a diagram, it *withs* in the other drawing object packages in order to use their operations. Over top of these autonomous packages are placed other packages to handle the inter-model and intra-model macro operations and object constraint management for the drawing model. Altogether, the drawing model packages total 14,000 lines of commented Ada source code.

I.3 Machine-independent Ada Graphical Support Environment

A machine-independent Ada graphical support environment (MAGSE) is needed to provide an interface between any window system and an Ada application. The MAGSE should sit on top of a window system, shielding the application from it while still supplying the application to perform graphical opertions. Why does Ada need a MAGSE like this? One reason is to contribute to the rather small amount of window and graphical support components written for Ada. A second reason is the intricacies and numerous subtle changes needed in an application when converting it from one kind of window system to another. A third reason is the amount of detail involved in creating a window with its many attributes and then performing event-checking on those windows. A fourth reason is the reliance of an application on a specific window system to supply sophisticated windows such as menus or dialog boxes. What makes the MAGSE machine-independent? The MAGSE is designed to be machine-independent in as far as Ada and the underlying window system are independent.

An object-oriented design approach to the MAGSE results in the creation of seven classes: drawing primitive, 2-D plane, 2-D matrix stack, 3-D pyramid, 3-D matrix stack, input device, and window manager.

The *drawing primitive* class contains lines, rectangles, circles, and text strings. It also has ds to draw and erase each of these primitives. One or more drawing primitives can be used nstruct complex drawing objects.

The *2-D plane* class contains a two-dimensional plane. It also has methods to set the plane's X and Y dimensions, and clip and render complex primitives in the plane. The *2-D matrix stack* class contains a stack for storing matrices which are used in performing two-dimensional transformations. It also has methods to push a matrix on the stack, pop a matrix off the stack, multiply another matrix times the matrix on the top of the stack, and perform two-dimensional rotate, scale, and translate operations on the top matrix of the stack.

The *3-D pyramid* class contains a three-dimensional perspective pyramid. It also has methods to set the X, Y, and Z dimensions of the pyramid, set the viewing location and viewing perspective of the pyramid, and clip and render complex primitives in the pyramid. The *3-D matrix stack* class contains a stack for storing matrices which are used in performing three-dimensional transformations. It also contains methods to push a matrix on the stack, pop a matrix off the stack, multiply another matrix times the matrix on the top of the stack, and perform three-dimensional rotate, scale, and translate operations on the top matrix of the stack.

The *input device* class contains a keyboard, a cursor and a 3-button mouse. It has methods to read the keyboard input, get the cursor position, detect mouse movement, and detect which mouse button was clicked. It also has methods to detect when a window event occurs with these windows.

The *window manager* class contains a window manager object. This object has methods to allocate and deallocate window storage and to retrieve a specific window from storage. It allocates drawing windows, acknowledge windows, confirm windows, dialog windows, column menu windows, sign windows, and text windows. The class has additional methods to create, display, hide, and destroy these windows.

The MAGSE design provides for it to be an interface between any window system and an Ada application. It sits on top of a window system, shielding the application from it and thereby creating a well-defined line of separation between the application and its graphical needs. A window system that needs such an interface is the X Window System. The MAGSE implementation serves as as Ada graphical interface to the xlib library of the X Window System.

The Massachusetts Institute of Technology (MIT) designed the X Window System with window mechanisms rather than window policy (25). Because of this intention, the X Window System can be used to mimic another window system while taking advantage of the networking capabilities of X. This generic window ability brings complications and complexity with it. To just create a window, establish its many graphical properties, and customize it to a certain style takes at least 21 X library function calls. In addition, ten of those calls return values or data structures that are needed as parameters to subsequent function calls that involve the window. The MAGSE captures all these complicated structures into a single record type and the numerous function calls into a few macro-functions. Some window flexibility is lost using the MAGSE, but much easier and simpler window handling is gained.

The object code for the window and graphics functions available in the X Window System are contained in a C-based library called xlib. Ada provides a compiler directive called a pragma interface which allows calls to C libraries to be made from Ada programs. Through the use of Ada pragma interface calls (Ada bindings) developed by SAIC (15), the MAGSE can access the X Window System's xlib and an Ada program can become an X Window client application. However, these Ada bindings are complicated to use. Once again the MAGSE comes through by offering a level of abstraction between an application program and the complexity of the xlib and the Ada bindings to the xlib . The MAGSE also provides specialized windows, such as menus and dialog boxes, that are not a part of the X Window System xlib. The aspiring Ada programmer doesn't have to spend days studying an xlib reference manual, examining sample X Window programs in C and Ada, and searching through the SAIC source code for parameter types and package names. He or she can just *with* the MAGSE interface into the application, and then declare variables and make subprogram calls as directed by the MAGSE interface specifications. By doing this, the application will become a genuine X client.

An application's view of the MAGSE is the MAGSE interface package. This package contains the package specifications for all the MAGSE package bodies. The MAGSE interface package contains no sign of complicated structures or numerous function calls to the X Window System or any other window system. All this detail is kept secure in the various package bodies. The application sees only a window identification type, choices of specialized windows, basic window manipulation routines, and basic drawing primitive routines. In the MAGSE package bodies, the internal structures, functions, and procedures are tailored to the specific window system that the MAGSE will be placed on top of. By implementing the MAGSE package bodies first for the X Window System, its structures and routines dictate mechanisms but no policy. The only restrictions on the window system are those imposed by the specialized windows offered by the MAGSE to an application. These windows (drawing, acknowledge, confirm, dialog, column menu, sign, and text) are not provided by the X Library (xlib) of the X Window System, but instead are implemented right in the MAGSE. Some of the ideas for implementing these specialized windows came from the X Window two-dimensional drawing tool developed by Cheng (5). Altogether, the MAGSE packages total 8,000 lines of commented Ada source code.

I.4 SAtoolII - an IDEF$_0$ Project Editor

The Ada source code that came from the ERD-to-OOD transformation of the IDEF$_0$ drawing model serves as a component along with the MAGSE in the implementation of a CASE tool called SAtoolII. SAtoolII is designed to be a combined IDEF$_0$ graphical project editor and data dictionary editor. It is only partially implemented at this time. The drawing model component provides the internal IDEF$_0$ structures for SAtoolII while the MAGSE provides the the graphical features and the specialized windows for the SAtoolII user interface. Because of its use of the MAGSE, SAtoolII takes advantage of the resources of the X Window System by utilizing the generic window and graphical features offered through the MAGSE interface. This keeps all X Window System library calls out of the actual SAtoolII application code.

I.5 Conclusions

This paper has provided solutions to the issues causing Ada to be shunned as a programming language for ERD modeling and graphical applications. It offered a six-step process for transforming an ERD model of a system into actual Ada source code and told about the use of this process in the transformation of the IDEF$_0$ drawing model. It also described the design of the machine-independent Ada graphical environment (MAGSE) and its implementation using Ada pragma interface calls to the C-based xlib library of the X Window System. In additon, it summarized how the the drawing model and the MAGSE serve as components in the partial implementation of a CASE tool called SAtoolII.

SAtoolII with its drawing model data structures, user interface, and graphical features stands as an example of a CASE tool that successfully incorporates entity-relationship modeling and graphics into an Ada application.

Bibliography

1. Austin, Kenneth A. and others. "An Entity Relationship Modeling Approach to IDEF$_0$ Syntax," *Proceedings of IEEE 1990 National Aerospace and Electronics Conference NAECON 1990*, *2*:641–645 (May 1990).
2. Barth, Paul S. "An Object-Oriented Approach to Graphical Interfaces," *ACM Transactions on Graphics*, *5*:142–172 (April 1986).
3. Booch, Grady. *Software Components with Ada*. Menlo Park, CA: Benjamin-Cummings, 1987.
4. Booch, Grady. *Object-Oriented Design with Applications*. Redwood City, CA: Benjamin-Cummings, 1991.
5. Cheng, WIlliam Chia-Wei. "tgif : Xlib-based 2-D Drawing Tool under X11." C computer software source code, 1990.
6. Connally, Ted D. *Common Database Interface for Heterogeneous Software Engineering Tools*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986 (AD-A189628).
7. Dodani, Mahesh H. and others. "Separation of Powers," *Byte*, *14*:255–262 (March 1989).
8. Dromey, Geoff. *Program Derivation: The Development of Programs from Specifications*. Sydney, Australia: Addison-Wesley, 1989.
9. Foley, James and others. *Computer Graphics: Principles and Practice (2nd Edition)*. Massachusetts: Addison-Wesley, 1990.
10. Foley, Jeffrey W. *Design of a Data Dictionary Editor in a Distributed Software Development Environment*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, June 1986 (AD-A172406).
11. Grudin, Jonathan. "The Case Against User Interface Consistency," *Communications of the ACM*, *32*:1164–1173 (October 1989).
12. Hartrum, Thomas C. "IDEF$_0$ Requirements Analysis." Class handout describing the use of IDEF$_0$ for software requirements analysis, October 1989.
13. Hartrum, Thomas C. *System Development Documentation Guidelines and Standards*, January 1989.
14. Hartrum, Thomas C. "Evaluation Form Comments for SAtool." Sorted comments on SAtool from student evaluation forms, January 1990.
15. Hyland, Stephen J. and Mark A. Nelson. "Ada Bindings to the X Window System." Ada computer software source code, 1987.
16. IBM. *AIX*. Technical Report, IBM, 1989. IBM AIX Marketing.
17. Johnson, Eric F. and Kevin Reichard. *X Window Applications Programming*. Oregon: MIS Press, 1989.
18. Johnson, Steven E. *A Graphics Editor for Structured Analysis with a Data Dictionary*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987 (AD-A190618).
19. Keim, Eric. "The KEYSTONE System Design Methodology," *Ada Letters*, *9*:101–108 (July/August 1989).

20. Kitchen, Terry L. *An Object-Oriented Design and Implementation for the IDEF$_0$ Esssential Data Model with an Ada Based Expert System*. MS thesis, AFIT/GCS/ENG/90D-07, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

21. Lewin, Stuart. "Ada Implementation of an X Window System Server," *Tri-Ada 1989 Proceedings*, pages 30–38 (1989).

22. Linnel, Dennis. "Graphical Interfaces Give Users Tools to Explore," *Government Computer News*, *9*:91–96 (September 1990).

23. Materials Laboratory, Air Force Wright Aeronautical Laboratories, Air Force Systems Command, Wright-Patterson AFB, OH 45433. *Integrated Computer-Aided Manufacturing (ICAM) Function Modeling Manual (IDEF$_0$)*, June 1981.

24. Myers, Brad A. "A Taxonomy of Window Manager User Interfaces," *IEEE Computer Graphics and Applications*, *8*:65–84 (September 1988).

25. Nye, Adrian and others. *Xlib Reference Manual for Version 11*. Massachusetts. O'Reilly and Associates, Inc, 1988.

26. Pountain, Dick. "The X Window System," *Byte*, *14*:353–360 (January 1989).

27. Ross, Douglas T. "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, *1*:16–34 (January 1977).

28. Sabella, Paolo and Ingrid Carlbom. "An Object-Oriented Approach to the Solid Modeling of Empirical Data," *IEEE Transactions on Computer Graphics and Applications*, *9*:24–35 (September 1989).

29. Scheifler, Robert W. and Jim Gettys. "The X Window System," *ACM Transactions on Graphics*, *5*:79–109 (April 1986).

30. Shlaer, Sally and Stephen J. Mellor. *Object-Oriented Systems Analysis : Modeling the World in Data*. New Jersey: Prentice-Hall, 1988.

31. Smith, Nealon F. *SAtool II: An IDEF$_0$ Syntax Data Manipulator and Graphics Editor*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989 (AD-A215289).

32. Sommerville, Ian. *Software Engineering*. Massachusetts: Addison-Wesley, 1989.

33. Sun Microsystems, Inc. *Sun View Programer's Guide*. Mountain View, CA, September 1986.

34. Thomas, Charles W. *An Automated/Interactive Software Engineering Tool to Generate Data Dictionaries*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1984 (AD-A152215).

35. Thomas, Dave. "What's in an Object?," *Byte*, *14*:231–240 (March 1989).

36. Urscheler, James W. *Design of a Requirement Analysis Design Tool Integrated with a Data Dictionary in a Distributed Software Development Environment*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986 (AD-A177663).

37. Wegner, Peter. "Learning the Language," *Byte*, *14*:245–253 (March 1989).

38. Young, Douglas. *X Window Systems: Programming and Applications with Xt*. New Jersey: Prentice Hall, 1989.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1990	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE AN ADA-BASED FRAMEWORK FOR AN IDEF$_0$ CASE TOOL USING THE X WINDOW SYSTEM	5. FUNDING NUMBERS
6. AUTHOR(S) Jay-Evan J. Tevis II, Capt, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/90D-15
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This thesis documents the design strategy and implementation methodology used in developing an Ada-based framework for SAtoolII using the X Window System. SAtoolII is designed to serve as an IDEF$_0$ graphical project editor and data dictionary editor. IDEF$_0$ is the ICAM Definition Method Zero graphical notation language adopted by the Air Force to produce a function model of a manufacturing system or environment. The Air Force Institute of Technology is conducting on-going research in the use of IDEF$_0$ in the requirements analysis phase of the software lifecycle. The thesis describes how SAtoolII was designed around an abstract entity-relationship model of the IDEF$_0$ language, an abstract model that was formulated in earlier research at AFIT into an essential model and a drawing model. It also describes the design of a machine-independent Ada graphical support environment which provides fundamental multi-window and graphical capabilities, while shielding an Ada application from the complexity of specific window systems. Following the design information, the thesis describes how the SAtoolII program was implemented incrementally, by developing an essential model component, drawing model component, machine-independent Ada graphics support environment, and graphical user interface. Plans exist to integrate these components in follow-on research.

14. SUBJECT TERMS Computer Aided Design, Software Engineering, Computer Graphics, Ada Programming Language, Object Oriented Design, X Window System	15. NUMBER OF PAGES 157
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL